# A Usability Study of Cryptographic API Design

Junwei Luo(✉), Xun Yi, Fengling Han, and Xuechao Yang

School of Computing Technologies, RMIT University, Melbourne, VIC 3000, Australia
{junwei.luo,xun.yi,fengling.han,xuechao.yang}@rmit.edu.au

**Abstract.** Software developers interact with cryptographic components via APIs provided by a cryptographic library to protect sensitive information such as passwords and files. While cryptographic algorithms have been standardised for over a decade, with variety of crypto libraries that implemented the algorithm, many developers struggle to use the library correctly. This paper evaluates 6 different cryptographic libraries written in 3 different programming languages to find out what factors affect usability. We analyse the usability of surveyed libraries with regards to its API call sequence, number of parameters, exception handling mechanism and documentation. In the end, several recommendations are provided to help developers choose which library to use and more importantly, this paper showcases a few common pitfalls for library designers to prevent common misuses when designing a cryptographic library.

**Keywords:** Cryptography · Usability analysis · Cryptographic APIs

## 1 Introduction

Cryptography is one of the most effective ways to protect private data in today's Internet, ranging from private and public key encryption schemes, message authentications, key exchanges, certificates and so on. Software developers employ different cryptographic components into their software to defend against malicious parties who attempt to compromise the software and steal sensitive information. Nowadays, most online services such as communications, bankings and others utilise different cryptographic components to strengthen the security and prevent attacks. While the practices for securing applications have been around for over a decade, many developers struggle to identify the correct way of deploying cryptography into their software. According to a survey [5] conducted by Egele et al., over three-fourth of surveyed application are vulnerable to various attacks, this is due to the fact that developers deployed cryptographic components into their software products incorrectly. Egele et al. refers such incorrect use of security-related components in production as cryptographic misuses.

Cryptographic misuses refer to incorrect implementations of cryptographic APIs in a product during the development process that potentially leads to security vulnerabilities. Incorrect use of cryptographic APIs, such as weak ciphers


X. Yuan et al. (Eds.): QShine 2021, LNICST 402, pp. 194–213, 2021.
https://doi.org/10.1007/978-3-030-91424-0_12

and key, can jeopardise the product and allow sensitive information to be stolen by adversaries. As a developer, it is essential to interact with APIs provided by the library to accomplish a specific task. While in many cases, APIs should be designed in a way that gives freedom to developers to achieve what they want to, it poses security issues as a poorly chosen parameter can potentially compromise the system entirely. Many developers found that using cryptography is challenging [14,15], due to the complexity of the cryptography and the design of the API being too complicating for non cryptographic experts to use it properly. Recent works have identified the problem of cryptographic misuses and its potential impacts to the industry, and come up with solution [3] that hides unnecessary complexity away from developers. While simplicity seems to improve the usability, no proper usability evaluation has been conducted as reports of cryptographic misuses continue to grow.

Therefore, it is necessary to conduct a series of case studies to evaluate the usability of different cryptographic APIs written in different languages. This paper will study 6 cryptographic libraries from 3 different programming languages to empirically evaluate their usability, a series of tasks is designed to simulate how cryptographic primitives are used by the developers. We evaluate symmetric encryption schemes provided by the library to study what factors hinder usability. While similar works [1,13,15] have been proposed previously that compares the usability of different cryptographic libraries, these works are specific to one particular programming language and did not discuss about API designs.

In the end, this paper aims to find out what affects the usability of a cryptographic library, and serves as a starting point for developers to choose which cryptographic library to use for their work. Moreover, we offer a list of recommendations for cryptographic library developers to improve the usability of libraries. To summarise, our contributions include the following:

1. We formally study the usability of cryptographic libraries written in 3 different languages, we choose 6 different cryptographic libraries to compare the usability.
2. We design a series of tasks to simulate the use of cryptography in reality and analyse the result in terms of their usability. How does the design of APIs affect the usability of a cryptographic library.
3. We compose a list of recommendations that help mitigate the issue when designing a new cryptographic library.

## 2 Related Work

Cryptographic misuses have become an issue for security researchers for more than a decade, this section covers the previous works for cryptographic misuses, such as detection mechanisms, mitigation mechanisms and the development of easy-to-use cryptographic libraries.

### 2.1 Evaluation of Cryptographic Misuses

Egele et al. [5] stated that over 11,000 applications downloaded from Google Play Store was surveyed, the team found that over 88% of them contained cryptographic misuses, ranging from the use of weak cipher, incorrect cipher settings and certificate verification. Fahl et al. [6] studied 13,500 applications on Android platform, and found that over 1000 app contained incorrect certificate verification code that can be exploited using Man-In-The-Middle (MITM) attack. Similar work [7] that analyses certificate verification on commercial grade development kit provided by company such as Amazon, also lacked usability due to API design being too confusing. Patnaik et al. [15] analysed over 2,400 questions related to cryptography on developer community such as Stack Overflow and found several common pitfalls that developers might face when using a crypto library, including poor documentation, lack of example code and bad API design with poorly chosen default settings. Similarly, [17] analysed around 1,500 reports from 5 different repositories and found that one fourth of the reports were posted due to the poor documentation as the developer cannot find appropriate answers from the document to accomplish the job, indicating that a well-written document can potentially improve usability of a library.

### 2.2 Mitigation Against Cryptographic Misuses

Approaches that utilise code analysis techniques to mitigate cryptographic misuses have also been proposed, CogniCrypt [10] is a code analysis framework that assists the use of cryptographic primitives in Java, CogniCrypt automatically scans for insecure cryptographic code and makes secure suggestions for developers to improve security. CogniCrypt also supports code generation based on use case scenarios chosen by the developers, these mechanisms mitigate the problem of crypto misuses. CrySL [11] is a definition language framework that allows developers to customise filtering rules for malicious code detection. CrySL allows the filtering rules to be exported as a template and shared across the community so that people can benefit from it without requiring deep understanding of writing complicating rules to detect code misuses. Other approaches such as CDRep [12] that enables automatic code detection and repair without accessing the source code for Android application have also been used extensively to mitigate the issue. CDRep is built on top of CRYPTOLINT [5] for its code detection, with added features to improve accuracy and reduce false-positive rate. As the source code of surveyed application is not available, decompilation is used to recover the code from Java Bytecode and patch incorrect cryptographic implementations. However, CDRep is limited to JCA only and has no support for third parties cryptographic libraries.

### 2.3 Toward Usable Cryptographic Libraries

The confusion around cryptographic misuses traces back to the creation of the library, with over complicating APIs and poor documentation, developers

found that using cryptography is challenging. Lots of efforts have been put into researching cryptographic libraries that focus on usability and simplicity. NaCl [3] is one of the earliest cryptographic implementation that adopts the idea of simplicity and usability by reducing the complexity and the number of secure choices that developers need to make in order to achieve security. To do that, NaCl removes the complicating API call sequence by wrapping several functions, such as key generation, cipher initialisation and encryption/decryption, in one function, where developers are not required to make secure choices about what cipher, mode of encryption and other parameters that are crucial to overall security. Since then, NaCl has inspired lots of researchers who work in designing usable cryptographic libraries, and many of its successors such as Libsodium and HACL* [18] are compatible with NaCl, with added primitives for signature and message authentication to provide more features, without requiring developers to change the code to adapt these libraries.

## 3 Preliminary

### 3.1 AES

Advanced Encryption Standard (AES) is a symmetric encryption specification established by NIST in 2001 [16], designed to replace its predecessor Data Encryption Standard (DES) published in 1977. AES is a block cipher where each block is 128 bits with different key lengths from 128, 192 to 256 bits. AES by itself is a block cipher that denote cryptographic transformation to scramble the data, and the mode of operation indicates how the block cipher is applied to the actual data for transformation. Depending on different cipher modes, additional parameters might be introduced to the cipher. Cipher Block Chaining (CBC) mode and Galois Counter Mode (GCM) are two commonly used modes for block ciphers. Both CBC and GCM mode utilise an extra parameter initialisation Vector (IV) to add randomisation to the ciphertext. Cryptographic randomisation refers to a mechanism where the message being encrypted more than once with the same cryptographic key results in different ciphertexts. This property is known as Indistinguishability of Chosen Plaintext Attack (IND-CPA). IND-CPA ensures that the probability of learning secret information from encrypted messages is negligible. CBC mode divides data into one or multiple blocks with a fixed size of 128 bits before the transformation, while the actual data might not meet such requirement. Therefore, padding is introduced to fulfil the block to ensure the last block is of the same size. Some CBC implementations might suffer padding oracle attack, where the adversary exploits the feedback mechanism of the padding implementation to deduce plaintext without having access to the key. On top of that, encrypted data in CBC mode is not authenticated, which means that the receiver cannot identify whether the ciphertext has been tampered by adversaries.

While mitigations have been proposed to address these attacks, CBC mode is later replaced by GCM mode, and is now considered obsolete in the latest TLS revisions due to its lack of message authentication. On the other hand, GCM is a counter mode that fixes these issues presented in CBC mode. GCM

requires two parameters called Nonce and counter, where Nonce is similar to IV to ensure the cipher is IND-CPA secure, and counter is used to append to the Nonce before the transformation. Difference between CBC and GCM is that CBC performs transformation on plaintext, whereas GCM transforms Nonce and the counter, and XOR the result with plaintext. Galois Message Authentication Code (GMAC) is the message authentication scheme in GCM that authenticates the ciphertext to ensure its validity.

## 4 Case Study

### 4.1 Task Design

To empirically evaluate the usability, a basic understanding of how developers usually use cryptography in their works needs to be established. A survey [14] indicates that over 30% of the questions related to cryptography posted on developer community is about symmetric encryption. This makes sense as many developers only need symmetric encryption for encrypting data such as a file or text.

We design two tasks that are common for developers, the first task is to generate a secret key corresponding to the cipher, followed by the second task: encrypt a text message with the secret key. Both tasks are designed to be simple to implement as most cryptographic libraries do support symmetric key cryptography. Our designed tasks did not involve in public key cryptography as these tend to be complicating for developers and more importantly, public key cryptography itself cannot provide security guarantee without the help of Certificate Authority to validate the identity of an entity and its related public key.

As for the experiment, we evaluate each library by completing the tasks described above and denote everything we found during the implementation of each task. On completion of implementing each task, we evaluate the usability based on common API design principles proposed by Green M [9] and Bloch J [4].

### 4.2 Language of Choices

C, Java and JavaScript are chosen for the experiment, as these languages have been widely used for many years. As for the cryptographic library itself, the following libraries are chosen. For C/C++, OpenSSL and Libsodium are chosen as the candidate, JCA, ACC and Tink are chosen as the candidate for Java, and lastly Crypto and SJCL are chosen as the candidate for JavaScript. OpenSSL is a commercial-grade cryptographic toolkit in C and C++ and is often considered the industry standard for applied cryptography as it provides a wide range of ciphers, along with other utilities such as key exchange, public key cryptograhy, message digest, X.509 and so on. OpenSSL is still in active development where more and more features are added to the repository and fixing security vulnerabilities. However, it is also infamous for its bad usability and poor document that bring confusion to the community and is difficult to use correctly. Libsodium is one of the successor of NaCl that brings modern features such as password hashing, key exchange and various hashing functions that are all missing from NaCl,

while maintaining backward compatibility with NaCl, as NaCl is no longer in active development.

For Java, Java Cryptography Architecture (JCA) is a built-in cryptographic toolkit introduced by Oracle. JCA offers different cryptographic primitives, message digest schemes, digital signature and so on. Apache Commons Crypto (ACC) is a lightweight cryptographic library that takes advantages of Intel AES-NI instruction to provide hardware acceleration for Java. The main advantage of ACC is to offer fast hardware AES transformation for better performance, its underlying cryptographic provider is a port of OpenSSL in C. Tink is a cross-platform cryptographic library introduced by Google, it claims to have better usability than the built-in JCA and have supports for modern cryptographic primitives that are missing from both JCA and ACC.

For JavaScript, Crypto is a cryptographic module from Node.js that covers a wide range of crypto primitives and utilities for hashing and certificate verification, Crypto in Node.js is essentially a port of OpenSSL into JavaScript. SJCL is a self-contained cryptographic library developed by a group of researchers at Stanford University and has supports for symmetric cipher, hashing and digital signature scheme.

Table 1 denotes the feature of each cryptographic library that we surveyed. While some libraries offer variety of ciphers, only the most commonly used ciphers are listed for simplicity. While some libraries only offer AEAD constructions, these implementations will be treated as if they were AE by ignoring the additional messages, as they are optional. For AES, Galois Counter Mode (GCM) is the mode of operation chosen to evaluate the libraries that support it.

**Table 1.** Features that are supported by corresponding cryptographic libraries that we chose for the experiment. •: requires input from developers. ○: requires no input from developers

| | Symmetric cipher | | | | | |
|---|---|---|---|---|---|---|
| | Key generation | Mode | Size | IV/Nonce | Default | Usability |
| OpenSSL | PRG | • | • | • | ? | |
| Libsodium | PRG | ○ | ○ | • | ? | ✓ |
| JCA | Keygen | • | • | • | AES ECB | |
| ACC | Keygen | • | • | • | ? | |
| Tink | Keygen | ○ | ○ | ○ | ? | ✓ |
| Crypto | PRG/KDF | • | • | • | ? | |
| SJCL | PRG/KDF | ○ | ○ | ○ | AES CCM | ✓ |

### 4.3 Evaluating the Solutions

We use the solutions as a starting point to find out the usability of these cryptographic libraries on the basis of the design principles proposed by Green M

[9] and Bloch J [4] for usable API designs. Among these design principles, we chose the following principles to evaluate the usability of these libraries as they made up the vast majority of cryptographic misuses and struggles found by other studies [14,15]. These principles are:

1. Make the APIs easy to use.
2. Make the APIs hard to misuses, with visible messages when used incorrectly.
3. Defaults should be safe and secure.
4. A well-written document with example code.

These usability principles are in line with a study on usability of cryptographic library [14], where the study found that over one-third of developers they surveyed struggled to identify the correct way of using cryptography, common obstacles include identifying the API call sequence, parameters required for the cipher and basic understanding of cryptographic implementation. The second design principle describes the error-handling mechanism of the library, some libraries do not give warning about potential security issues, such as incorrect key size and mismatched tag. The third design principle suggests that a secure default value should always be preferred if developers did not specify about what value to use. Lastly, documentation of the library also plays a critical rule to improve usability as it provides an official guideline for developers to look for features, tutorials and so on, a good API design without documentation might introduce confusion to developers, which could potentially lead to insecure implementation of a cryptographic component.

To evaluate each principle, the following criteria will be applied:

**Design Principle 1:** Usability of APIs is evaluated based on whether or not the APIs can address the common obstacles listed in [14]. These obstacles are: API call sequence, identifying parameters and understanding API implementation. To do that, we evaluate the solution based on the number of lines of code and parameters. Logical Lines of code (LLoC) is used to measure API call sequence, whereas the Number of Parameters (NoP) is used to measure the complexity of a cryptographic API and its implementation. For symmetric encryption, the minimum required parameters should be at least 2, which will be the key and message respectively, the more parameters involved, the less usable the library is considered to be. The exact same principle will be applied to other tasks as suggested in [15] that the library designers should make an effort to minimise the number of choices that developers have to make.

**Design Principle 2:** We pay our attention to the exception handling mechanism of each library. Specifically, how does the library respond to common cryptographic misuses such as weak cipher, incorrect parameters and key size as these issues make up most of the common misuses [5]. The team also suggests that the issues could have been mitigated if either the compiler or cryptographic providers sent out warnings about the potential misuses.

**Design Principle 3:** We investigate default configuration used by the library. Egele et al. [5] stated that over 50% of surveyed applications relied on the default

cipher provided by cryptographic providers. We analyse the following three questions: Does the API provide a default option, If so, what is the default cipher provided by the API? Is the default cipher secure?

**Design Principle 4:** We evaluate the usability of a document in a cryptographic library. Documentation plays a critical rule [4] in usability as it provides a standard guideline for developers to get started. To do that, the following questions will be asked to each library: Does the library include a document? Is the desired function easy to locate? Did the document record all exported functions, methods, parameters and so on? Does the library provide example code for common use cases to mitigate development?

## 5 Study Results

AES has become industry standard for symmetric encryption, this paper will focus on analysing AES implementation of each library, all libraries support AES as part of their encryption engines. We choose GCM as the mode of operation for evaluating AES as it is considered the most widely used mode among the others. Although Libsodium has supports for AES-GCM, its supports are platform-dependent as it takes advantages of hardware-assisted AES instructions for security reason. Thus the evaluation excludes Libsodium.

### 5.1 API Call Sequence

API call sequence or initialisation sequence refers to the code that requires to be executed in sequence to achieve any functionality. A study [14] found that over one-third of surveyed targets believe that such API call sequence is difficult to identify, due to the lack of cryptographic background and practice. To find out if a library outperforms others with regards to the first design principle, Logical Lines of Code (LLoC) is used to measure the amount of work required for implementing a feature.

**Key Generation.** The first task of evaluation is to generate the symmetric key corresponding to the cipher. In this case, a symmetric key with the length of 16 bytes. Key generation typically falls into these categories: Pseudorandom Generator (PRG), Key Generator (KeyGen) or Key Derivation Functions (KDFs). All 6 libraries have a relatively simple interface for key generation, regardless of which key generation method was implemented aforementioned.

However, the differences between these libraries emerge on the second task, which is to encrypt a message using the key from task 1. To better understand call sequence, we divided it into three stages: Initialisation, Update and Finalisation.

**Init.** Libraries with no usability designs tend to require prior knowledge of initialisation sequence in order to initialise it properly. OpenSSL scores the lowest due to the fact that developers are responsible for identifying the API call

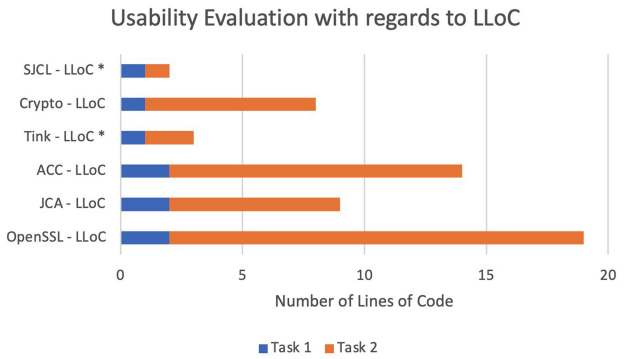


**Fig. 1.** Assessment criteria based on the number Logical Lines of Code (LLoC), LLoC describes the amount of efforts to complete a task. *: Usability claims

sequence and memory allocation. Followed by the ACC in Java, which also has similar pattern and requires input from developers. While JCA shares many similarity with ACC as they are both written in Java, JCA facilitates the initialisation sequence by making a parameter optional. Specifically, JCA has a default crypto provider where developers can choose from to achieve better performance or functionality whereas ACC explicitly requires it in order to initialise the cipher. Tink uses a template to store settings for different cipher, thus the need of manually initialising the cipher no longer exists. Lastly, Crypto in Node.js is also similar with regards to initialisation, while facilitating usability by adopting language-specific features such as loosely typed objects, anonymous functions and so on. Finally, SJCL does not require initialisation to be done.

**Update.** Cryptographic update is rather straightforward once the initialisation is done, as most mistakes occur during the initialisation. To achieve that, developers make one function call to inform the library to perform cryptographic transformation. While different libraries handle the interaction differently, we observe two patterns during the update. Specifically, how data is exchanged between the user and the library. Both OpenSSL and ACC require an extra parameter in the update function for receiving the ciphertext, whereas other libraries take advantage of the return instruction to send back ciphertexts to the user.

**Final.** Lastly, finalisation verifies ciphertext and generates authentication tag as an extra layer of security, as GCM mode computes the auth tag by computing the GMAC of the ciphertext for tampering prevention. OpenSSL and Crypto require developers to make two function calls to compute and retrieve the tag respectively. On the other, JCA and ACC only require one function call and the tag will be appended to the end of ciphertexts. Libraries with usability claims eliminates the need of finalisation as they are done during the update stage.

Figure 1 denotes the corresponding LLoC for both tasks across all libraries with AES supports. It is clear that initialisation stage usually results in higher number of LLoC due to the additional codes for setting up, while this is beneficial for experts who understand cryptography in practices, it exposes too much risk for those who do not have cryptographic background.

### 5.2 Identifying Required Parameters

**Key Generation.** The first task involves the generation of a cryptographic key, which is commonly done using either PRG, KeyGen or KDF. In most cases, KeyGen is implemented as a wrapper of PRG with the length of the key pre-filled to the PRG. As different ciphers might require keys in different sizes, having a KeyGen function as supposed to a general PRG might improve the usability as developers do not need to know the correct key length. Among 6 libraries, half of these libraries, namely OpenSSL, Crypto and SJCL use PRG for key generation, whereas KeyGen is utilised in the other half of the libraries for key generation. The result is obvious, all PRG functions require input of key length from developers in order to generate the key for encryption, whereas KeyGens simply return the newly generated key without requiring input from developers.

**Init.** Similar to call sequence where we categorise the process into stages, we divide the process of analysing parameters into different stages for better comparison. Typically the parameter needed includes the following: cipher, mode, key, Initialisation Vector (IV) or nonce, plaintext, ciphertext, and auth tag. During the initialisation, cipher, mode, IV or nonce are typically required before the encryption can occur, whereas plaintext and ciphertext are required in update and finally auth tag is retrieved in finalisation.

During initialisation, OpenSSL introduces an optional parameter that can be used for switching to a different crypto engine, similar to the crypto provider in JCA and ACC. ACC scores the lowest among other 5 libraries due to the fact that developers are required to explicitly set up crypto provider for the library to realise the backend used for actual encryption. Apart from the crypto provider which is required to be set explicitly, ACC and JCA share lots of similarities as parameters necessary for both libraries to set up are identical. Tink uses a template with pre-defined settings for every primitive supported by the library to facilitate the initialisation. On the other, Crypto requires the length of the auth tag to be set explicitly, the only library we surveyed requires this, and the rest is identical to the others with regards to the number of required parameters. Finally, SJCL only requires the key to be set whereas others are optional and the library will handle it if it is not explicitly set.

**Update.** As for the update, we observe there exists several pattern across different libraries for handling actual cryptographic transformation. OpenSSL requires both plaintext and ciphertext to be passed as parameters to the function that executes the transformation. On top of that, the size of the plaintext is also

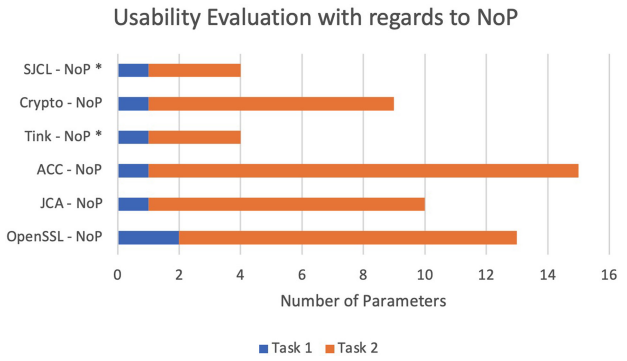


**Fig. 2.** Number of Parameters (NoP) used for completing task 1 and 2. NoP describes the minimum numbers of parameters required for completing a task. *: Usability claims

required for the function to work correctly. In Java, ACC follows the same pattern as the OpenSSL, it is not surprising since ACC is essentially a wrapper of OpenSSL, and the team kept the same workflow as its backend to minimise discrepancy. JCA on the other hand, eliminates such complexity and only the plaintext is required for the cipher object to perform encryption, and the ciphertext is returned upon function completion. Similarly, Tink, Crypto and SJCL follow the same idea where only the plaintext is needed for the job to be done.

**Finalisation.** To handle authentication tag, users of OpenSSL require to allocate free memory for holding the tag, as OpenSSL does not use return instruction to facilitate usability. In Java, ACC complicates the operation by requiring lots of parameters for computing the tag. It is surprising that 5 parameters are involved in the process of finalisation and more importantly, none of the parameters are actually required as they are integers such as the length of the plaintext and ciphertext, offsets and so on. As an OO language, we believe it could have been done better like JCA where it requires no input from developers and be able to compute the tag automatically. Tink on the other does not require involvement of the end users for finalisation as it was done in update stage. Lastly for the library in JavaScript, Crypto simplifies the process by making one function call with no parameter and the auth tag can be retrieved. SJCL follows the same design pattern as Tink and does not require any action to be taken for retrieving the tag. Figure 2 concludes the usability of API sequence with regards to the number of parameters for surveyed libraries.

### 5.3 Exception Handling for Incorrect Use

Exception handling describes the mechanism that informs developers about any potential issues that might arise due to incorrect inputs. In applied cryptography,

common cryptographic misuses such as weak cipher, incorrect key and parameters, are found in many applications [5]. Most libraries do not have a exception handling mechanism for key generation, or the mechanism is not documented. OpenSSL exceptionally provides a function for inspecting the last error. Some key generation functions in these libraries are OS-dependent. For example, a common source of pseudorandom generator is the system device in Unix-like OS called */dev/urandom*. Such device is not available in earlier version of the OS, which might cause problems generating a cryptographically secure values that can be used as a symmetric key.

For the second task, OpenSSL once again scores the lowest among other 5 libraries due to its lack of secure checks for weak ciphers, incorrect key length and related parameters. OpenSSL developers state that the user should be responsible for ensuring the correctness of all parameters. Crypto in Node.js and SJCL score similarly where both libraries are capable of identifying incorrect length of key, while ignoring other checks such as weak ciphers and incorrect parameters that are also critical to the security. JCA and ACC also have similar results where both libraries are able to identify not only the incorrect length of key, but also the incorrect size of parameters. Lastly, Tink is perhaps the best out of 6 libraries, as we have no way of mistakenly generating incorrect parameters due to its design that relies on pre-defined templates, which denote the initialisation sequence, generations of key, corresponding parameters and cipher objects.

Overall, none of the library that supports for weak ciphers has a mechanism to remind developers about the use of weak ciphers, albeit this feature is fairly trivial to implement. JCA and ACC send warnings to some extent, Crypto in Node.js and SJCL are similar regarding to the error handling mechanism as well. In the meantime, Tink does not suffer from potential incorrect use due to its pre-defined template that automatically takes care of everything.

Table 2 denotes the finding in exception handling mechanisms from libraries that support AES. Ideally, the developers of a cryptographic library should either ensure that these issues are not present, or to design an exception handling system that tells the users about any potential issues that might lead to security incidents if it is left without proper handling. Libraries that have one or more criteria labelled as "N/A" are considered a pass as the criteria is not applicable to the library.

### 5.4 Default Cipher

Some libraries offer default values if developers do not specify explicitly, while such feature is beneficial, some default options are considered insecure while the default option is still being used extensively [5] and the developers are unaware of the situation which can lead to security incidents. As task 1 is not applicable for the criteria, we pay our attention to the task 2, which uses newly generated key to encrypt a message.

**Table 2.** Exception handling of each library, sorted by the type of warning. ✓: indicates that the library is capable of identifying the potential issues. ✕: indicates that the library is not capable of identifying potential issues. N/A: Criteria is not applicable to the library.

| | Weak cipher | Incorrect key size | Incorrect parameters | Constant parameters |
|---|---|---|---|---|
| OpenSSL | ✕ | ✕ | ✕ | ✕ |
| JCA | ✕ | ✓ | ✓ | ✕ |
| ACC | ✕ | ✓ | ✓ | ✕ |
| Tink | N/A | N/A | N/A | N/A |
| Crypto | ✕ | ✓ | ✕ | ✕ |
| SJCL | N/A | ✓ | ✕ | ✕ |

Among 6 libraries, JCA allows a cipher object to be retrieved by specifying the string "AES" during the initialisation. SunJCE provider, which is by default the cryptographic provider in JCA, returns "AES/ECB/PKCS5Padding" to the developers. Electronic Code Block (ECB) mode is insecure due to the lack of the following: ECB is not IND-CPA secure, while it might consider secure if the key is used only once and the size of message is less than or equal to exact one block, it is not feasible as plaintexts are typically larger than 16 bytes in most cases. On top of that, ECB lacks randomness, which explains why the key can only be used once per message of size 16 bytes. Absence of authentication code for ciphertext in ECB mode makes it difficult for a receiver to identify if the ciphertext is valid. JCA reference guide states that ECB is the easiest mode to use, whereas it should be avoided if the message is larger than a single block. A study [5] pointed out that around 40% of all surveyed applications written in Java used the string "AES" to retrieve cipher object. It is unclear as to how such design was adopted in SunJCE, but it is clear that it makes up a portion of weak cipher misuses.

Another library that offers default cipher is SJCL in JavaScript, it returns CCM mode as the default mode of operation. CCM mode is a counter mode with combination of CBC-MAC for message authentication. CBC-MAC is deterministic and uses AES with CBC mode to compute Message Authentication Code with a fixed Initialisation Vector. While such mode is provably secure and part of the latest TLS revision, its performance is not comparable to GCM mode as CBC-MAC is not parallelisable. On top of that, AES-CCM is a MAC-then-encrypt diagram [2], which has no guarantee of ciphertext integrity until it is decrypted and more importantly, CCM mode has a lower adoption rates compared to GCM mode which has been widely used for years.

As the third design principle evaluates the default cipher and its security, only the library with supports for secure default cipher receives a pass. In the case of libraries with AES supports, SJCL in JavaScript passes the test while others fail to do so, due to either insecure default cipher or no supports for default cipher.

### 5.5 Documentation

The fourth design principle describes the importance of documentation in usability. To evaluate, the following questions are asked: Is the document available for a library? Is the desired function easy to locate? Did the document denote exported functions and explain the functions and parameters? Did the document have a example code for all exported functions. As documentation is available for all 6 libraries, we move our attention to evaluate the second and third criteria using task 1 and 2.

Task 1 involves the generation of a symmetric key, OpenSSL scores the lowest among others, as its documentation is very confusing, with all functions exported in one place without categorising the methods based on functionalities. Searching for a keyword "random" results in over 25 matches and only a handful of functions are what the task 1 is after. With all the confusion among locating the desired function to use, explanation of the function is clear and easy to understand, example code of this function is not available, or it might be available but not in the same page, we consider this a fail as developers should not be redirected to other places looking for the same function.

Compared to OpenSSL, JCA and ACC have a relatively simple document where it categories functions based on functionalities, while trying to explain everything in one page, both JCA and ACC fail to provide example codes for key generation, where the example code in JCA for key generation was not made for AES, a minor modification of that example code can make it compatible with AES, given the developers had prior knowledge of identifying the place to change. ACC's key generation example is absent but is documented.

Crypto in Node.js meets all 4 criteria, where every function exported is easy to locate, and receives a clear explanation and example code. For other two libraries with usability claims, Tink offers a document where the example code can be used with little modification, due to its usability designs where the interface is minimal. On top of that, Tink offers several key management mechanisms where a key can be exported as a file, or uploading to offshore Key Management System (KMS) run by third parties. Lastly, SJCL has a document where it explains the process of key generation with PBKDF2 and PRG. PBKDF2 derives a symmetric key from a passphrase by appending a random salt to HMAC and iterates multiple times to reduce the possibility of attacks. It is unclear how many iterations are used for PBKDF2 as it is directly related to the security, and the documentation of PBKDF2 is absent as of the time of writing this survey. Albeit an alternative solution namely PRG is also available and documented.

Task 2 uses the key generated in task 1 to encrypt a message. OpenSSL once again scores the lowest due to the envelope design that standardises the interface for different crypto primitives. By searching "aes" in the document, it returns the name of the cipher used in the EVP cipher routines, where the EVP cipher routines are the functions responsible for the actual cryptographic transformation. Example code for symmetric encryption is available inside a function that initialises the envelope, though the example code is not made for AES, the code can be used as to demonstrate the initialisation sequence of the

cipher. The inconsistency across OpenSSL's document leads to many confusions in the community, as denoted by a study [15] where OpenSSL is considered the worst among other surveyed libraries regarding its documentation.

Documentation for Symmetric encryption in JCA is largely facilitated as the document is categorised by features, the provided example code presents the usage of GCM mode to encrypt data. ACC on the other hand, offers several example codes for the use of AES, while the example code is made for CBC mode only. As ACC is a port of OpenSSL in C, it requires the length of the ciphertext to be known for the cipher to operate. Internally, it uses the last 16 bytes to store authenticated tag, while data before the last 16 bytes are considered ciphertext. Decryption might fail if the array that stores the encrypted data contains empty bytes in between the auth tag and ciphertext, this is difficult to debug as such behaviour is not documented, the cipher will throw a warning about the tag being mismatched, as it mistakenly includes the empty bytes as part of ciphertext and computes the auth tag.

Crypto in Node.js provides example code in the document that demonstrates the use of CBC mode with scrypt as KDF to derive a 192 bits key from a passphrase. scrypt is a KDF that is designed to be computationally intensive to prevent against hardware attacks. While it is recommended to use KDF to derive a cryptographic key from a passphrase with low entropy, the example code fails to clarify some important aspects of using KDF and its parameters. In the example code, a fixed salt is applied to the KDF, whereas it is recommended to ensure the uniqueness of each salt and only use it with a unique password. The example code also fails to clarify that scrypt is memory intensive, which might be problematic if the application is deployed in an environment with limited memory. Although the document mentions about the uniqueness of salt, notices should be given at all places where the scrypt is used as developers might not be aware of the issue if the example code works.

Tink documents features such as symmetric encryptions with example code to facilitate the development, while it has supports for multiple symmetric encryption schemes, the name of these templates (used in Tink in order to initialise cipher object) is difficult to locate, as they are only written in the API documentation. SJCL on the other hand, while the mode of encryption is optional, lists the supported modes in the example code. Similar to Crypto in Node.js, SJCL provides PBKDF2 as the KDF for deriving passphrases. However, SJCL also fails to clarity important aspects of using PBKDF2, namely its iterations. PBKDF2 is computationally intensive in terms of computational time, whereas scrypt is computationally intensive in terms of memory usage, which means that the number of iterations is directly related to the cost of computation. By default, the iteration is set to 10,000, which was considered enough back then when the library is introduced to the public in 2009, it is recommended to use iterations over 50,000 or even 100,000 as the computational power of modern hardware continues to increase. On top of that, no documentation about changing the iteration is found in the page where the example code for symmetric crypto is presented.

## 6 Discussion and Recommendation

While it is true that OpenSSL is undeniably the industry standard for all cryptographic algorithms, results show that it is by far the worst among other 5 libraries with regards to usability. On top of that, the lack of detailed explanation in its official document and its cumbersome error handling mechanism result in many confusions to the community. Experiments show that users require to manually check for the correctness of each function call, one can completely bypass checks for authenticated tag by ignoring return value from **EVP_DecryptFinal**. It is unclear why developers of OpenSSL have decided to make plaintexts available to the user, given the fact that the authenticity of the text might be susceptible due to the failure of verifying the tag. After encryption, users are responsible for handling ciphertext, Initialisation Vector (IV) and authentication tag for storage or network transmission as they are the minimum required for decryption. While it is true that one can bypass checks for authentication tag in OpenSSL, such check is mandatory in most other libraries, and cannot be bypassed in such a way as OpenSSL did.

Although OpenSSL is not suitable for people who are unfamiliar with applied cryptography, it does offer greater flexibility and functionality for different security purposes. It is unfortunate that there are only a few libraries written in C that offer greater usability, and for these libraries, only one of them has supports for AES and this feature is limited to supported processors only due to several security considerations around AES.

Unlike C programming language which does not have an official cryptographic library built-in to the standard SDK, Java on the other hand receives an official cryptographic framework (JCA). JCA has quickly become the default cryptographic library in Java, due to its popularity and part of the standard Java SDK. It was also the default crypto library for Android until 2018 when Google has replaced it with its own cryptographic providers. One biggest issue with JCA is the default option provided by SunJCE provider when using AES. In JCA, a cipher object can be retrieved by calling crypto factory with a string parameter indicating what cipher to use. To facilitate usability, SunJCE allows cipher name to be simplified, which is where the issue occurs. By default, it uses the pattern similar to **AES/GCM/NoPadding** for the crypto factory to realise what cipher and its mode to use. However, one might simply pass in a simplified string **AES** and the crypto provider will response with **AES/ECB/PKCS5Padding**, which is the least secure mode among all other modes for AES. This simplified crypto name makes up over one-third of common cryptographic misuses in Java [5]. It remains unclear as to why such decision is made in the first place, the development team behind JCA clearly understands the issue as they also inform users about potential risks of using such simplified string in JCA documentation. One thing for sure is that many users did not read the warning as it has become one of the most common crypto misuses in JCA.

ACC uses designs similar to JCA where users might choose which crypto provider to use, and ACC is backed by OpenSSL to take advantages of hardware-assisted AES encryption for better performance and security. While the design

of ACC is similar to JCA in terms of API call sequence, it requires more parameters to handle cryptographic operation. More specifically, it is similar to what OpenSSL does, with regards to the number of parameters required for **Update** and **Final**. It is less common for Java to handle low level operations such as manipulating byte array in a similar fashion to C. Such drawback is mitigated by using ByteBuffer object, a facilitated way to manage binary data, and both JCA and ACC support it. Inconsistency between encryption and decryption in ACC is also found. Particularly, it is found in the **Update** method during encryption where it takes as input plaintext, size and offset value and produces ciphertext stored in a given byte array at the starting offset value. During decryption, **Update** can be eliminated and one can call **Final** to complete both decryption and verification. It is worth noting that by default, ACC treats the last 16 bytes as authentication tag and everything above is ciphertext. A successful decryption requires both the size of ciphertext and its offset in the byte array, if a byte array contains information other than the ciphertext and authentication tag.

Lastly, Tink is a cryptographic library that offers implementations from different languages, such as Java and Python, developed by a group of security researchers at Google. The purpose of Tink is to reduce the possible cryptographic misuses that could potentially result in a security incident. In our experiments, Tink does seem to improve usability quite a lot, a concept of key template is introduced by Tink, which denotes settings such as key size, IV and other parameters that one usually finds when initialising a crypto object. A typical crypto implementation requires inputs about crypto, mode, key and other parameters from developers, whereas Tink hides such hassle by wrapping all settings necessary within one object. A crypo object can then be retrieved on basis of the key template, which defines what cipher to use and its parameters. With Tink, the only way to use its underlying crypto is to specify a key template, and the library handles the rest of them including generation of keys and IVs, setting up crypto engine and so on. Upon completing initialising the crypto engine, one simply calls encrypt to retrieve ciphertext and decrypt to verify the tag and reveal plaintext. As both IV and tag are appended to the ciphertext itself, it facilitates both storage and network transmission as users are no longer required to come up with a protocol to handle transmission over the network and disassemble the chunk of data for decryption.

In regards to JavaScript, a language that has gained attention in recent years, we analyse two libraries, Crypto Node.js has a relatively simple interface and is backed by OpenSSL in C. Therefore, both Crypto Node.js and OpenSSL are somewhat similar with regards to API call sequence, while Crypto Node.js has a much clearer interface due to adoption of language-specific features from JavaScript to simplify the work. Although at the end of encryption users also get ciphertext, IV and tag, one can serialise data easily as demonstrated in the official document and is recommended for users to follow the instruction to reduce unnecessary misuses.

SJCL is the last candidate among all 6 libraries and the design shares a lot of similarity with Tink. In particular, it eliminates the need of initialising cipher

object due to the fact that the library supports AES only. On top of that, SJCL implements AES-CCM as the default cipher and all additional parameters will be generated if not explicitly specified. One notable feature is that SJCL implements PBKDF2 for deriving passphrase keys, as supposed to the keys generated at random and more importantly. In many cases, Key Derivation Function (KDF) is more preferable as it is difficult for users to memorise continuous random bytes.

**Recommendation.** In C, OpenSSL is still the must-go option for better versatility. Otherwise, NaCl or Libsodium are better choices than OpenSSL with regards to usability as they were designed specifically to mitigate such issues. For Java users, JCA guarantees its compatibility across different platforms and architectures, whereas Tink should offer a better experience due to its user-centred design that facilitates cryptographic usages. For JavaScript, Crypto module in Node.js is a better choice for versatility, whereas SJCL offers a clean user interface, along with better compatibility across different browsers.

As C does not have a exception handling mechanism built-in to the language itself, it is perhaps worth conducting researches on how a better error handling system can be designed, as supposed to manually check for error code by the end users. We demonstrate that OpenSSL does not have a rigorous error handling mechanism, when it is misused, one might continue to decrypt a file without knowing its validity due to the way OpenSSL was designed.

On the other hand, both Java and javaScript offer a better exception handling scheme, as the scheme has always been a part of the design of the language. Tink's implementation shows that cryptographic library can be easy for people without cryptographic background to use it properly, without worrying about potential misuses.

Furthermore, we give recommendations for library designers to prevent misuses. We observe that many libraries require two function calls for initialisation to be done, whereas libraries with usability claims reduce that to one function call. We suggest that initialisation can be simplified as many libraries have achieved that goal without issues. Same goes for finalisation where the auth tag is computed and attached to the ciphertext. While some might argue that there are reasons for not combining the tag to the ciphertext as they should be stored in different places in some cases, we recommend that the library can optionally include an interface that simplifies the process, while retaining the original design for advanced users.

For the second principle, libraries should take advantage of built-in exception handling mechanism wherever possible. Although many libraries utilise that to report issues to the end users, the error message is rather unintuitive as developers could not make sense of the message produced by the library when errors occur [8,15]. For languages that have no built-in supports for error handling, we believe that more researches can be done to investigate a better way to overcome such limitation.

For the third principle, we discourage the use of default values, this is because it could either break backward compatibility or open up a security vulnerability,

whenever the default value becomes insecure. To mitigate the issue, we suggest that libraries can optionally introduce a mitigation tool that allows one to migrate to another cipher without introducing too many breaking changes.

## 7 Conclusion

This paper presents a study about the usability of cryptographic APIs from 6 libraries written in 3 programming languages. We set up a series of tasks as to simulate what developers use cryptography and analyse the usability with regards of their API call sequence, parameters, exception handling mechanisms and documentation. Experiments show that many libraries are designed under an assumption that developers understood cryptography to some extent, whereas the situation is the exact opposite. Many cryptographic misuses arise because of bad API designs. Libraries such as Tink and SJCL do seem to improve usability as users are no longer required to understand cryptography before using it correctly. Given the presence of issues with regards to cryptographic issues, further research is needed to improve API designs, documentation and the way that libraries interact with users as these factors affect usability. In our future work, we will focus on analysing libraries in a source code level, to have a better understanding of how decisions were made to the library, and categories these findings as to provide a guideline for cryptographic researchers when designing a cryptographic library.

## References

1. Acar, Y., et al.: Comparing the Usability of Cryptographic APIs. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 154–171. IEEE (2017)
2. Bellare, M., Namprempre, C.: Authenticated encryption: relations among notions and analysis of the generic composition paradigm. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 531–545. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44448-3_41
3. Bernstein, D.J., Lange, T., Schwabe, P.: The security impact of a new cryptographic library. In: Hevia, A., Neven, G. (eds.) LATINCRYPT 2012. LNCS, vol. 7533, pp. 159–176. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33481-8_9
4. Bloch, J.: How to design a good API and why it matters. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, pp. 506–507 (2006)
5. Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An empirical study of cryptographic misuse in android applications. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 73–84 (2013)
6. Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., Smith, M.: Why eve and mallory love android: an analysis of android SSL (in) security. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 50–61 (2012)

7. Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., Shmatikov, V.: The most dangerous code in the world: validating SSL certificates in non-browser software. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 38–49 (2012)
8. Gorski, P.L., et al.: Developers deserve security warnings, too: on the effect of integrated security advice on cryptographic {API} misuse. In: Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018), pp. 265–281 (2018)
9. Green, M., Smith, M.: Developers are not the enemy!: the need for usable security APIs. IEEE Secur. Priv. **14**(5), 40–46 (2016)
10. Krüger, S., et al.: CogniCrypt: supporting developers in using cryptography. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 931–936. IEEE (2017)
11. Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M.: CrySL: an extensible approach to validating the correct usage of cryptographic APIs. IEEE Trans. Softw. Eng. (2019)
12. Ma, S., Lo, D., Li, T., Deng, R.H.: CDRep: automatic repair of cryptographic misuses in android applications. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, pp. 711–722 (2016)
13. Mindermann, K., Keck, P., Wagner, S.: How usable are rust cryptography APIs? In: 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 143–154. IEEE (2018)
14. Nadi, S., Krüger, S., Mezini, M., Bodden, E.: Jumping through hoops: why do java developers struggle with cryptography APIs? In: Proceedings of the 38th International Conference on Software Engineering, pp. 935–946 (2016)
15. Patnaik, N., Hallett, J., Rashid, A.: Usability smells: an analysis of developers' struggle with crypto libraries. In: Fifteenth Symposium on Usable Privacy and Security ({SOUPS} 2019) (2019)
16. Standard, N.F.: Announcing the advanced encryption standard (AES). Federal Information Processing Standards Publication **197**(1–51), 3–3 (2001)
17. Zibran, M.F., Eishita, F.Z., Roy, C.K.: Useful, but usable? Factors affecting the usability of APIs. In: 2011 18th Working Conference on Reverse Engineering, pp. 151–155. IEEE (2011)
18. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: a verified modern cryptographic library. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1789–1806 (2017)