

Mitigating Cryptographic Mistakes by Design

Maximilian Blochberger

Tom Petersen

Hannes Federrath

University of Hamburg

{lastname}@informatik.uni-hamburg.de

ABSTRACT

Developers struggle to integrate cryptographic functionality into their applications. Many mistakes have been identified by related work and tools have been developed for detecting, automatically repairing, or otherwise assisting developers in secure integration of cryptographic functionality. We present a cryptographic API that has been designed to prevent cryptographic mistakes for developers without a background in cryptography. For that purpose, common cryptographic mistakes were categorized systematically. A qualitative user study was performed that evaluates the usability of the API. The results indicate that a simple, comprehensive API can aid developers in implementing cryptographic functionality securely without much effort.

KEYWORDS

Cryptographic mistakes, API design, API usability

1 INTRODUCTION

Results from Egele et al. [17] and Nadi et al. [39] show that the average developer makes severe mistakes when using cryptography in their products, which puts their users at risk. Developers usually have no security background and lack resources to acquire the required knowledge [8, 16]. Findings from related work regarding developer behavior indicate that app development is performed with minimum cost and effort [9]. Therefore, adding cryptographic functionality should be made possible with low cost and effort. Green and Smith [24] also argue that developers should be supported by offering convenient cryptographic APIs. In addition, developers can be nudged into developing more privacy-friendly applications, if they are offered simple APIs that fit their use case [29]. Transferring these findings to the misuse of cryptographic libraries, it is reasonable to assume

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

MuC'19 Workshops, Hamburg, Deutschland

© Proceedings of the Mensch und Computer 2019 Workshop on Usable Security und Privacy Copyright held by the owner/author(s).

<https://doi.org/10.18420/muc2019-ws-302-02>

that presenting simple, intuitive cryptographic APIs to developers results in fewer mistakes when they use cryptography in their software.

For that purpose, we implemented a convenient API for different scenarios, where cryptographic functionality is utilized. In order to protect against common cryptographic mistakes, we first present a systematization derived from related work and give an overview of research on this topic. Second, we discuss design choices that our cryptographic API is based on. Third, we discuss evaluation methods and provide results of a qualitative user study.

2 RELATED WORK

The field of research on cryptographic misuse is mostly composed of techniques that are used to detect [16, 17, 32, 33, 38, 41, 42] or repair [36, 41] cryptographic mistakes, discussions of possible implications [25, 34], and recommendations on how to integrate cryptographic functions securely [7, 14, 16, 17, 32, 34, 41].

Only a few scientific contributions describe how a cryptographic API can be designed in a way that it prevents mistakes in the first place [10, 11, 20]. However, there are some cryptographic libraries available that claim to be easy to use and misuse resistant. There is Sodium¹, which in turn is based on NaCl², introduced by Bernstein et al. [10]. Sodium is used as a basis for our implementation presented in Section 4. Monocypher³ and Themis⁴ offer similar APIs to Sodium. Tink⁵ has a different API, but requires somewhat knowledgeable developers, due to the exposure of low-level primitives.

Related work, that detects or repairs cryptographic mistakes, analyzes Android apps [14, 17, 38, 42], iOS apps [35], and projects on GitHub [16]. The methodology used for detecting cryptographic mistakes can be divided into static and dynamic analysis – well-known techniques for analyzing software: (i) *Static Analysis*, which is used to inspect the source code or binaries of applications without executing them. This is done by either automatically [17, 38] or manually [14] extracting and evaluating features from application

¹<https://libsodium.org>

²<https://nacl.cr.yp.to>

³<https://monocypher.org>

⁴<https://www.cossacklabs.com/themis/>

⁵<https://github.com/google/tink>

resources. (ii) *Dynamic Analysis*, which is used to inspect the behavior of applications during run time. This is usually done manually [14, 35, 42] as it depends on the user interacting with the application. Muslukhov et al. [38] performed source attribution of cryptographic mistakes found in Android applications and found that most mistakes are included via third-party libraries.

Arzt et al. [7], Iacono and Gorski [26], and Nadi et al. [39] identify commonly required use-cases depending on cryptography that were used as a basis for use-cases provided in the API we have implemented.

3 SYSTEMATIZATION OF CRYPTOGRAPHIC MISTAKES

We analyzed selected papers and categorized cryptographic mistakes based on their cause. Some papers [14, 16, 42] already present a categorization of the mistakes they discuss. This was taken into account as we saw fit. Mistakes, for which the cryptographic engineer is clearly responsible were left out, such as implementation bugs within the cryptographic library. These mistakes can not be prevented by developers, even if they utilize the cryptographic API as intended. Mistakes can be assigned to multiple categories and categories are not independent of each other, e. g., the lack of examples in the documentation (*knowledge base*) contributes to higher *usage complexity*, as the developer has to figure out, how to use the API correctly. The remaining part of this section introduces the categories our systematization consists of.

Initialization

Initializing cryptographic keys, passwords, pseudorandom number generator (PRNG) seeds, or nonces – including initialization vectors (IVs) – incorrectly might render the cryptographic operation ineffective. Since there are many possible mistakes during initialization, this category was further divided into the following subcategories.

Predictable Sequences. Most initialization mistakes use or lead to predictable sequences, where an attacker might be able to, e. g., predict a cryptographic secret. Using a random number generator that is not cryptographically secure [19, 34] or even using a cryptographically secure random number generator seeded with low entropy [14, 17, 25, 34] results in predictable sequences. Predictability is not only a problem for cryptographic secrets, but also for observable values, such as nonces [14, 17, 35, 36, 38, 42].

Re-use. Re-using values is a special case of *predictable sequences*, where the security is weakened by using values more than once. This can even be problematic, if values are generated securely. This category includes the re-use of

keys [14], nonces [16, 20, 42], or PRNG seeds [14, 34]. Common cases for re-use are static values, i. e., values that are constant, which includes static keys [14, 17, 25, 35, 36, 38, 41], PRNG seeds [14, 19, 36, 38, 41], CTR counters [14, 16], and nonces [14, 16, 17, 19, 35, 41] – with the special case of a nonce being constantly zero [16, 17, 22, 35]. Hard-coded values are static values as well, with the additional property that they can easily be retrieved from the application itself. Hard-coding is mentioned for keys [14, 34, 35, 42], passwords [42] – some of them used for password-based encryption (PBE) [14], and nonces [14].

Weak Values. Another special case of *predictable sequences* are weak values. For example, if cryptographic secrets are too short, they can easily be broken using a brute-force attack. Therefore, it is a mistake to have short keys [14, 19, 34, 42]. Some cryptographic algorithms have the requirement that cryptographic keys have a certain structure in order to provide the intended security [19, 34]. If values are constantly zero, they are considered weak as well, specifically mentioned are nonces [16, 17, 22, 35].

Source. Nonces are supposed to be truly random, but some developers derive them from the key [34] or from the message [14]. Since true randomness is hard to achieve, some developers derive randomness from seemingly unpredictable sources, such as the key itself [25] or from files [25].

Insecure Defaults

Values, which are not explicitly set by the caller are sometimes initialized with default values provided by the cryptographic library. This is usually a good thing, as this can reduce the complexity of the API. However, some default values are insecure [13, 14, 16, 17, 20, 22, 33, 39, 41]. For example, some libraries use ECB as the default mode of operation such as the Java cryptographic API (JCA) [14, 17, 33, 41] or PyCrypto [16]. The latter also uses a static IV as default for AES in CBC mode [16, 22].

Weak Algorithms

Many cryptographic algorithms have been proved insecure over time. Although they are outdated, they are often still supported by cryptographic libraries due to backward compatibility. However, since developers lack the knowledge, which algorithms to use, they might not adopt newer and more secure algorithms. An example, which we already mentioned, is the usage of ECB [14, 17, 19, 22, 25, 33, 36, 38, 41, 42]. Related work mentions DES [14, 17, 19, 22, 32, 33, 34, 38, 42] and RC4 [14, 19, 34, 38] as insecure algorithms for symmetric encryption. MD4 [22, 34, 42], MD5 [17, 19, 22, 33, 34, 36, 42], and SHA1 [14, 19, 33, 34] are specifically mentioned as insecure hash functions. Other cases of weak algorithms are self-implemented primitives or protocols, e. g.,

a re-implementation of AES [14] or authenticated encryption [16]. The usage of deterministic encryption (not IND-CPA secure) [35] is considered weak as well.

Validation

Even if encryption is performed securely, mistakes can be made during validation of certificates or other assets such as ciphertexts. This topic is prevalent in related work about TLS, regarding the validation of TLS certificates, where many things can go wrong [18, 42], such as not validating at all, missing or erroneous host name validation, accepting expired or revoked certificates, and validating only a subset of the whole certificate chain. We added lack of authenticated encryption [20] and the usage of expired cryptographic keys [42] to this category.

Persistence of Secrets

The security of cryptographic mechanisms depends on the used secrets being actually kept secret. If the key or password is stored in plain along the ciphertext, the cryptographic mechanism is rendered ineffective. There are two major purposes for which cryptographic secrets are persisted: authentication and re-use. Secrets kept for authentication can be secured more easily as only a salted hash should be stored. Secrets for re-use however need to be stored reversibly, which might result in keys stored in plain [25].

Usage Complexity

Many mistakes can be attributed to complex API calls, where the correct parameters or call sequences need to be respected in order for the mechanism to be secure [13, 16, 21, 39]. The user is faced with many choices obscure to him, due to the lack of cryptographic background. In addition, some cryptographic libraries add feature bloat and support very rare protocols or outdated algorithms. Examples for insecure parameters chosen for specific algorithms are RSA without OEAP [14, 25, 31, 42], CBC with PKCS5 padding [14, 25], and PBE without salt [14, 17, 25] or with less than 1000 iterations [14, 17, 36, 38, 40, 41] as recommended in RFC 2898 [30]. Further mistakes can be attributed to insecure combinations of cryptographic primitives or the lack of necessary steps, such as forgetting to initialize a nonce with a random value [16, 42]. Missing but required features like authenticated encryption forces developers to provide their own custom and insecure combinations [16]. Non-obvious error handling [16, 25], the use of uninitialized values that leads to *predictable sequences* [34], as well as missing compile and run time validations can be attributed to the lack of modern programming language features [16]. For deeper insights into how developers struggle with the JCA specifically, refer to Nadi et al. [39]. Jaeger and Levillain [28] discuss the intrinsic security characteristics of programming languages.

```

1 private static byte[] encrypt(byte[] raw, byte[] clear)
  ↳ throws Exception {
2     SecretKeySpec skeySpec = new SecretKeySpec(raw, "AES");
3     Cipher cipher = Cipher.getInstance("AES");
4     cipher.init(Cipher.ENCRYPT_MODE, skeySpec);
5     byte[] encrypted = cipher.doFinal(clear);
6     return encrypted;
7 }
8
9 byte[] keyStart = "this is a key".getBytes();
10 KeyGenerator kgen = KeyGenerator.getInstance("AES");
11 SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
12 sr.setSeed(keyStart);
13 kgen.init(128, sr); // 192 and 256 bits may not be available
14 SecretKey skey = kgen.generateKey();
15 byte[] key = skey.getEncoded();
16
17 byte[] encryptedData = encrypt(key, b);
18 byte[] decryptedData = decrypt(key, encryptedData);

```

Listing 1: Insecure example taken from Stack Overflow, showing how to encrypt data with the JCA

Knowledge Base

In addition to the usage complexity described earlier, developers have a hard time finding resources helping them to figure out how to integrate cryptographic software into their applications securely. It is hard to find useful information in the official documentation of cryptographic libraries [16, 39], officially provided examples are insecure [16] or the examples lack a security discussion [16, 17, 22]. Third-party information sources, such as Stack Overflow, are also filled with insecure suggestions and examples [4, 5, 19].

Other

There are other mistakes that were considered by related work, such as not using cryptography at all [14], side-channels introduced through compiler optimizations [34], or incomplete processing of internal buffers or clearing secrets from memory [14, 27, 33].

Example

The following scenario visualizes the problem of cryptographic misuse in practice. Assume a developer wants to add encryption to an Android application. Since he has no background in cryptography, he simply searches for “android encryption example” and finds a highly rated accepted answer on Stack Overflow [1]. Even though there is a warning message and several comments indicating that the example is insecure, this might not prevent the developer from using the code. There are seven actual cryptographic mistakes and two potential weaknesses in the code snippet displayed in Listing 1, which was taken directly from the cited Stack Overflow answer.

- (i) In Line 3 "AES" is specified as the cipher that should be used. The block mode and padding are not provided and will default to "AES/ECB/PKCS5PADDING", even though this problem in JCA was already pointed out in 2013 [17] (*insecure default*), and
- (ii) a discussion regarding ECB being the default was not found in the documentation of the JCA (*knowledge base*).
- (iii) The algorithm choices (Line 2, 3, 10, and 11) are string values and typos will result in run time errors. As described, the string can also be used to configure the block mode and padding. Knowledge about which algorithms are available and considered secure is required (*usage complexity*).
- (iv) The variable `keyStart`, which is used as a PRNG seed in Line 12, is initialized with a hard-coded value in Line 9 (*initialization*) [19].
- (v) The PRNG uses outdated SHA1, as can be seen in Line 11 (*weak algorithms*).
- (vi) In Line 1–15 a custom key derivation function is implemented (*weak algorithms*).
- (vii) The ciphertext `encryptedData` is deterministic (*weak algorithms*), and
- (viii) not authenticated (*validation*).
- (ix) The usage of non-specific types for the `encrypt()` function allows for swapping both arguments accidentally. This could result in the clear text being used as key (*initialization*).

This example illustrates that the problem of cryptographic misuse is pervasive. The approach we take to prevent as many mistakes as possible is to carefully design the cryptographic API. We will cover this design in the following section.

Limitations

The systematization can only be considered work-in-progress. It lacks a clear and reproducible paper selection process, as well as a sound methodical review, such as presented by Maass et al. [37]. The categories were created by a single person and hence are subjective. Categorization should be the result of inter-coder agreement and has to be performed with at least two persons. Aside from these limitations, many cryptographic mistakes were extracted from related work, providing a coarse overview of the topic.

4 DESIGN CHOICES

Bloch [12] formulated characteristics for good APIs. Most notably for the following section, "APIs should be easy to use and hard to misuse", they "must coexist peacefully with the

platform" (*platform integration*), and "Fail fast" ("Compile-time is best"). Based on these characteristics and the categories of cryptographic mistakes identified in the previous section, we designed a cryptographic API for the Swift programming language. In order to make the API available to developers, we implemented an open source framework for iOS and macOS called Tafelsalz⁶.

Swift was chosen due to its modern language features, and the author's familiarity with it and the iOS and macOS platforms. However, our design choices are conceptual and can be applied to other languages, such as Java, or the more modern Kotlin programming language, which are available on Android, as well. Ideally, a usable cryptographic API is provided by the platform vendor as part of the standard SDK or adopted by other cryptographic libraries [23].

Tafelsalz is based on Sodium, which already protects against many mistakes, such as *initialization*, *insecure defaults*, *weak algorithms*, and *validation* [16]. The *usage complexity* varies, as there are various bindings for different programming languages that offer slightly different APIs than the C-based library `libsodium`. The API differences are a necessity, due to *platform integration* requirement.

Initialization mistakes, e. g., length checks, are handled at run time in `libsodium`. In contrast to C, modern programming languages like Swift have more strict type checking mechanisms, typed null pointers (optionals), overflow protection, other compile time checks, and more features that can be utilized to move errors from run time to compile time. This allows failing before a cryptographic function is called (*fail fast*). The type (or `class`) should prevent objects from being instantiated, if they are invalid, e. g., due to an unexpected size. Typing can further support developers to initialize the internal values of objects. Ideally, developers do not need to know that a cryptographic key has to be initialized with a cryptographically secure random number. The cryptographic engineer can define the correct behavior in the type's constructor already. This does not prevent developers from making mistakes, as objects often need to be initialized with a given value, such as a persisted or shared key. However, this is another use-case and should require a different API call.

Another example is the *persistence of secrets*, which is hard to get right. Sodium only offers persistence of secrets used for authentication, by transforming the secret into a string that can be stored in a database or similar. Storing secrets for reuse however, is a platform-, even device-specific problem and thus not offered by Sodium. The security of the cryptographic operation depends on the secrecy of the key. If the key is persisted, its secrecy depends on the security of the storage medium. Ideally, a hardware authentication device should

⁶<https://github.com/blochberger/Tafelsalz>

```

let alice = Persona(uniqueName: "Alice")
let box = SecretBox(persona: alice)!
let plaintext = "Hello, World!".utf8Bytes
let ciphertext = box.encrypt(plaintext: plaintext)
let decrypted = box.decrypt(ciphertext: ciphertext)!

```

Listing 2: Symmetric encryption with Tafelsalz

```

// Alice's side
let alice = Persona(uniqueName: "Alice")
let bob = Contact(publicKey: bobsPublicKey)!
let box = SecretBox(persona: alice)!
let plaintext = "Hello, World!".utf8Bytes
let ciphertext = box.encrypt(message: plaintext, to: bob)

// Bob's side
let bob = Persona(uniqueName: "Bob")
let alice = Contact(publicKey: alicesPublicKey)!
let box = SecretBox(persona: bob)!
let decrypted = box.decrypt(message: ciphertext, from:
↳ alice)!

```

Listing 3: Asymmetric encryption with Tafelsalz

be used. However, these are not always available. On iOS and macOS Tafelsalz utilizes the Keychain, which is secured by the tamper-resistant co-processor called Secure Enclave on iOS devices and some recent Macs. On devices without Secure Enclave, the Keychain is secured with PBE [6]. This makes the API *harder to misuse*, as the developer does not have to implement a custom persistence mechanism.

The *knowledge base* of Tafelsalz consists of use-case oriented examples and detailed API documentation. The documentation includes warnings for dangerous operations, e. g., a warning indicates that initializing a cryptographic key with a given byte sequence should not be used for generating new cryptographic keys.

In addition, Tafelsalz offers a more use-case oriented API, so that the developer finds an interface for the specific task he intends to accomplish instead of primitives he must combine to achieve this. The use-cases resulted from the usage of Sodium, functionality that often contains cryptographic mistakes, and the implementation of various demonstrators. The following subsections provide insights into improvements Tafelsalz offers compared to Sodium to prevent further cryptographic mistakes.

Identity Management

There are two types of identities (or actors) in a classic cryptographic setting:

Persona An identity, of which the user possesses the secret key, so that he can en- or decrypt messages for this persona at will. The term *persona* was chosen, as it refers to “the personality that a person projects in

```

let password = Password("Correct Horse Battery Staple")!
let hashedPassword = password.hash()

// Store `hashedPassword.string` to database.

// Authenticate a user with the password he entered
if hashedPassword.isVerified(by: enteredPassword) {
    // The user is authenticated successfully.
}

```

Listing 4: Password hashing with Tafelsalz

public” [2], which in contrast to *identity* infers that a single person can have multiple secret keys, e. g., for different contexts such as work and private life.

Contact An identity, of which the user only possesses a public key, so that he can encrypt messages for the contact or decrypt messages received from the contact.

A persona is identified by a unique name (per app) and can be used as a source for cryptographic secrets for symmetric or asymmetric encryption. If the developer wants to encrypt a string that only the persona is supposed to know, he can utilize the `SecretBox` class as depicted in Listing 2. Symmetric encryption with the default secure algorithm provided by Sodium will be performed. The persona object checks whether a persisted secret already exists in the system’s Keychain that can be used for symmetric encryption. If none exists, it will create a new one. Secrets are app-specific, so that personas with the same name in two different applications are segregated. If the developer wants to asymmetrically encrypt a message from Alice to Bob, he can use a slightly similar call, as depicted in Listing 3.

Password Hashing

Passwords that are used to authenticate users must not be stored. A salted password hash should be stored instead. In order to make the API usable, the developer does not need to know which algorithm is used, and he does not need to figure out how to correctly salt passwords. Hence, a convenient password hashing function has been added, which can be used as depicted in Listing 4.

Internal Protection

In addition to the presented API, the Tafelsalz framework protects secrets and passwords that are kept in memory from being accessed. This functionality is provided by Sodium as well, however the developer needs to actively invoke the appropriate functions. Since Tafelsalz uses typed objects, protections can be enabled as part of the type’s functionality. To mitigate timing attacks, passwords and hashes are compared in constant-time automatically.

Discussion

The implementation of the usable cryptographic API Tafelsalz can be used by developers without security background to integrate cryptographic functionality without much effort. However, the API lacks general applicability and is tailored to specific use-cases.

The API is designed in a way that developers without security background can use it properly without having to change default values. For developers with advanced background, some configuration is possible. In addition, Sodium lacks backward compatibility, as it removes *weak algorithms*. Data encrypted or hashed with deprecated algorithms from a previous version of the library has to be updated with more secure alternatives. Similarly, developers might be required to use specific algorithms that are not included in Sodium, which is not possible with Tafelsalz as well.

Note that not every cryptographic mistake identified in Section 3 is addressed in our design. We included at least one mistake from each category. In addition, more findings from the field of usable API design should be taken into account.

The documentation was only done superficially and best practices should be taken into account in the future. Future evaluation should also determine whether the class and function names are appropriate, as the expected lack of security background within the target audience might cause confusion for terms like `password.hash()` or `SecretBox`, which were adopted from Sodium.

A final minor finding: We implemented a demonstrator that supports interoperation between iOS and Android. We found that using the sensitive configuration for the password hashing algorithm of Sodium exceeds the default memory limits for Java Virtual Machines on Android. If this level of protection is required, the memory limits can be increased.

5 EVALUATION

Several qualitative as well as quantitative methods can be used to evaluate the usability and comprehensibility of Tafelsalz and its documentation. In this section we will describe the evaluation steps we have taken so far and elaborate on possibilities for future evaluation.

There are two major settings in which Tafelsalz or similar libraries can be evaluated: First by observing or interviewing developers during their daily work when using Tafelsalz to perform a task and second by providing specific tasks for the developers to solve using our library. We will focus on the second setting since it enables us to evaluate specific aspects of the library, like the quality of the documentation or the cryptographic primitives being used.

Introductory Challenge

We have developed an introductory challenge to work with Tafelsalz called DCrypt⁷, which serves two main purposes: (i) act as a starting point for developers willing to learn the concepts being employed in Tafelsalz, and (ii) give us the opportunity to qualitatively evaluate the usability of the Tafelsalz interface and its documentation.

DCrypt provides an application stub, so that the developer can concentrate on the relevant missing cryptographic operations. The challenge consists of several tasks with increasing difficulty, which are described abstractly. This is on purpose to force the developer to contact the documentation and understand the interface Tafelsalz provides.

The first task is to implement encryption and decryption with the Tafelsalz framework. Participants have to contact the documentation and can implement their encryption based on an example in the documentation. The second task is to add unit tests for their implementation. The unit tests allow the participants to work more efficiently, especially when implementing further tasks. This reduces the overall time required for the challenge. In addition, participants can reflect on their ideas, especially if they solved the first task by copying example code from the documentation. The third task is to test, whether decryption still works after they relaunch their application and to fix it in case it does not. The goal of this task is to introduce the problem of persisting keys. The last task is to encrypt files in a way, that they can be decrypted by another group. This introduces the problem of sharing or transferring keys, and the expected solution is to use PBE.

It is planned to include further tasks dealing with key exchange, key derivation, hashing, and asymmetric encryption.

Pilot Study

We had the chance to present Tafelsalz at the *GI DevCamp 2018*⁸, a workshop for developers. The workshop consisted of different topics that were presented to all participants in the beginning. Our pilot study was one of the topics. After the introduction of the topics, the participants could choose which topic they want to attend. 12 students with different backgrounds in computer science participated in our pilot study, where they had to solve all tasks of our introductory challenge. All of them but one had no or just marginal knowledge in cryptography and the use of cryptographic APIs.

After a short introduction to the problem to be solved, we confronted them with the challenge and let them work on it on their own just with the help of the Tafelsalz documentation. This process was observed by two hosts, which were also available for questions. Afterwards we presented

⁷<https://github.com/AppPETs/DCrypt>

⁸<https://hamburg.dev-camp.com>

an example for possible cryptographic mistakes when using other libraries like the one given in Section 3 and had a focus-group-like discussion about the participants experiences. On one hand, the observation during the challenge as well as the discussion allowed us to evaluate our design. On the other hand, the presentation allowed us to educate the participants by increasing the awareness of the problem of cryptographic mistakes. That way participants gained something from taking part in our workshop.

We just want to give some high-level results of our observations and the discussion here. The participants despite their missing knowledge of cryptography had no problems with successfully implementing secure symmetric encryption functionality in DCrypt (apart from programming language related questions). Only the concept of storing keys in the macOS Keychain in combination with the Tafelsalz persona concept lead to some misunderstandings which could not completely be solved by just contacting the documentation. With some hints provided by the hosts the participants were able to solve all challenges successfully within a time frame of about 1.25 h. About 0.75 h were used for introducing the problem, presenting common cryptographic mistakes, and discussing the challenges.

To end this section we just want to present a student's initial statement on how he would proceed when being confronted with a task like the one given in DCrypt:

“Well, you visit Stack Overflow and take the first hit for your search term.”

This is one reason, why we should continue to make our cryptographic APIs as foolproof as possible.

Limitations

We have just taken the first steps for the evaluation of Tafelsalz. Our results from the aforementioned workshop look promising in regard to the usability of our library for developers with minor cryptographic knowledge. We got feedback which shows some shortcomings in the library as well as its documentation that we can use to improve their quality. Further evaluation steps in the future should include well-conceived tests with a larger test population and reproducible quantitative results. In addition, our API should be compared to other existing cryptographic libraries in order to determine whether our design choices are effective.

Future Work

Several methods are known from the fields of psychology or UX testing [15], which can be employed to get insights into strengths and shortcomings of Tafelsalz. These include:

Questionnaires Asking developers who use the library for their opinions on Tafelsalz. An obvious problem is that there is no possibility to interact with the developers.

Expert interviews Asking experts in the field of cryptographic APIs for their opinion on the design and implementation of Tafelsalz.

Moderated testing Interacting with and observing the developer, when he performs a task. Optionally the so-called think-aloud method can be used, where the developer is asked to tell his thoughts during the task. This can give further insights into problems during working with Tafelsalz.

Focus groups Moderated discussion with several participants. Open questions and the influence of different participants can lead to ideas, which would not come up with other methods.

Another way of evaluating whether Tafelsalz succeeds in empowering developers without a background in cryptography to write secure cryptographic code, is to compare it to other cryptographic libraries [3]. This requires a carefully selected setting for the to be accomplished task. The compared libraries must support the required primitives and it has to be ensured that possible caveats, like complicated procedures when including the libraries for the used development environments, affect the test results as little as possible.

Furthermore, the possibly different preexisting knowledge has to be taken into account. A developer with some understanding of cryptographic principles might perform completely different in comparison to a complete novice in the field of cryptography. To avoid knowledge effects, each developer could execute the task with each library in a random order (to prevent distorting learning effects) or at least the developer's background has to be considered during evaluation. The comparison can use several objective metrics:

- Could the task be accomplished at all?
- Were any cryptographic mistakes made in the task?
- How much time was needed to accomplish the task?

Additional questionnaires might lead to further insights about the subjective experiences the developers had while working with Tafelsalz.

6 CONCLUSION

In this paper we have presented a systematization of cryptographic mistakes, details about our cryptographic API, which tries to avoid cryptographic mistakes by design, and insights into its evaluation. However, our contributions have some potential for future work. The discussed limitations show that further work on this topic is required to reduce the amount of cryptographic mistakes in today's applications. We show that designing usable and comprehensible cryptographic APIs enables developers without security background to secure their users' data.

ACKNOWLEDGMENTS

This research was supported by the Federal Ministry of Education and Research of Germany (BMBF) as part of the AppPETS project⁹. Thanks to the anonymous reviewers for their valuable feedback.

REFERENCES

- [1] [n. d.] Retrieved May 22, 2019 from <https://stackoverflow.com/a/6788456/5082444>.
- [2] [n. d.] Retrieved July 1, 2019 from <https://www.merriam-webster.com/dictionary/persona>.
- [3] Yasemin Acar et al. 2017. Comparing the usability of cryptographic apis. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society.
- [4] Yasemin Acar et al. 2017. How internet resources might be helping you develop faster but less securely. *IEEE Security & Privacy*, 15, 2.
- [5] Yasemin Acar et al. 2016. You get where you're looking for: the impact of information sources on code security. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society.
- [6] Apple Inc. 2019. iOS Security – iOS 12.3. White paper. https://www.apple.com/business/docs/iOS_Security_Guide.pdf.
- [7] Steven Arzt et al. 2015. Towards secure integration of cryptographic software. In *Onward!* ACM.
- [8] Rebecca Balebako et al. 2014. The privacy and security behaviors of smartphone app developers. In *USEC*.
- [9] Birgitta Bergvall-Kåreborn and Debra Howcroft. 2013. The future's bright, the future's mobile: a study of apple and google mobile application developers. *Work, Employment & Society*, 27.
- [10] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2012. The security impact of a new cryptographic library. In *LATINCRYPT (LNCS)*. Volume 7533. Springer.
- [11] Daniel J. Bernstein et al. 2014. Tweetnacl: A crypto library in 100 tweets. In *LATINCRYPT (LNCS)*. Volume 8895. Springer.
- [12] Joshua J. Bloch. 2006. How to design a good API and why it matters. In *OOPSLA Companion*. ACM.
- [13] Kelsey Cairns and Graham Steel. 2014. Developer-resistant cryptography. In *STRINT*. W3C/IAB.
- [14] Alexia Chatzikonstantinou et al. 2015. Evaluation of cryptography usage in android applications. In *BICT*. ICST/ACM.
- [15] John W. Creswell. 2014. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications.
- [16] Somak Das et al. 2014. IV=0 Security: Cryptographic Misuse of Libraries. Technical report.
- [17] Manuel Egele et al. 2013. An empirical study of cryptographic misuse in android applications. In *ACM Conference on Computer and Communications Security*. ACM.
- [18] Sascha Fahl et al. 2012. Why eve and mallory love android: an analysis of android SSL (in)security. In *ACM Conference on Computer and Communications Security*. ACM.
- [19] Felix Fischer et al. 2017. Stack overflow considered harmful? the impact of copy&paste on android application security. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society.
- [20] Christian Forler, Stefan Lucks, and Jakob Wenzel. 2012. Designing the API for a cryptographic library - A misuse-resistant application programming interface. In *Ada-Europe (LNCS)*. Volume 7308. Springer.
- [21] Martin Georgiev et al. 2012. The most dangerous code in the world: validating SSL certificates in non-browser software. In *ACM Conference on Computer and Communications Security*. ACM.
- [22] Peter Leo Gorski et al. 2018. Developers deserve security warnings, too: on the effect of integrated security advice on cryptographic API misuse. In *SOUPS @ USENIX Security Symposium*. USENIX Association.
- [23] Matthew Green. 2012. The anatomy of a bad idea. Retrieved July 1, 2019 from <https://blog.cryptographyengineering.com/2012/12/28/the-anatomy-of-bad-idea/>.
- [24] Matthew Green and Matthew Smith. 2016. Developers are not the enemy!: the need for usable security apis. *IEEE Security & Privacy*, 14, 5.
- [25] Peter Gutmann. 2002. Lessons learned in implementing and deploying crypto software. In *USENIX Security Symposium*. USENIX.
- [26] Luigi Lo Iacono and Peter Leo Gorski. 2017. I do and i understand. not yet true for security apis. so sad.
- [27] ISE. [n. d.] Password managers: under the hood of secrets management. Retrieved February 19, 2019 from <https://www.securityevaluators.com/casestudies/password-manager-hacking/>.
- [28] Éric Jaeger and Olivier Levillain. 2014. Mind your language(s): A discussion about languages and security. In *IEEE Symposium on Security and Privacy Workshops*. IEEE Computer Society.
- [29] Shubham Jain and Janne Lindqvist. 2014. Should i protect you? understanding developers' behavior to privacy-preserving apis. In.
- [30] Burt Kaliski. 2000. PKCS #5: password-based cryptography specification version 2.0. *RFC*, 2898.
- [31] Jeffrey Knockel, Thomas Ristenpart, and Jedidiah R. Crandall. 2018. When textbook RSA is used to protect the privacy of hundreds of millions of users. *CoRR*, abs/1802.03367.
- [32] Stefan Krüger et al. 2017. Cognicrypt: supporting developers in using cryptography. In *ASE*. IEEE Computer Society.
- [33] Stefan Krüger et al. 2018. Crysl: an extensible approach to validating the correct usage of cryptographic apis. In *ECOOP (LIPICs)*. Volume 109. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.
- [34] David Lazar et al. 2014. Why does cryptographic software fail?: a case study and open problems. In *APSys*. ACM.
- [35] Yong Li et al. 2014. Icryptotracer: dynamic analysis on misuse of cryptography functions in ios applications. In *NSS (LNCS)*. Volume 8792. Springer.
- [36] Siqi Ma et al. 2016. Cdrep: automatic repair of cryptographic misuses in android applications. In *AsiaCCS*. ACM.
- [37] Michael Maass et al. 2016. A systematic analysis of the science of sandboxing. *PeerJ Computer Science*, 2.
- [38] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. 2018. Source attribution of cryptographic API misuse in android applications. In *AsiaCCS*. ACM.
- [39] Sarah Nadi et al. 2016. Jumping through hoops: why do java developers struggle with cryptography apis? In *ICSE*. ACM.
- [40] Alena Naiakshina et al. 2017. Why do developers get password storage wrong?: A qualitative usability study. In *ACM Conference on Computer and Communications Security*. ACM.
- [41] Duc-Cuong Nguyen et al. 2017. A stitch in time: supporting android developers in writingsecure code. In *ACM Conference on Computer and Communications Security*. ACM.
- [42] Shuai Shao et al. 2014. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *DASC*. IEEE Computer Society.

⁹<https://app-pets.org>