API Usability of Stateful Signature Schemes

Alexander Zeier(✉), Alexander Wiesmaier, and Andreas Heinemann

Darmstadt University of Applied Sciences,
Haardtring 100, 64295 Darmstadt, Germany
{alexander.zeier,alexander.wiesmaier,andreas.heinemann}@h-da.de

Abstract. The rise of quantum computers poses a threat to asymmetric cryptographic schemes. With their continuing development, schemes such as DSA or ECDSA are likely to be broken in a few years' time. We therefore must begin to consider the use of different algorithms that would be able to withstand powerful quantum computers. Among the considered algorithms are hash-based signature schemes, some of which, including XMSS, are stateful. In comparison to stateless algorithms, these stateful schemes pose additional implementation challenges for developers, regarding error-free usage and integration into IT systems. As the correct use of cryptographic algorithms is the foundation of a secure IT system, mastering these challenges is essential.

This work proposes an easy-to-use API design for stateful signature schemes, using XMSS(MT) as an example. Our design is based on findings from literature as well as on a series of interviews with software developers. It has been prototypically implemented and evaluated in small-scale user-studies. Our results show that the API can manage the stateful keys in a way that is transparent to the user. Furthermore, a preliminary online-study has shown that the API's documentation and applicability are comprehensible. However, due to the transparent state management, many of the study's participants were unaware of using a stateful scheme. This might lead to possible obstacles. Our current API design will serve as the basis for a larger user-study in order to review our preliminary findings in the next step.

Keywords: Post-quantum cryptography · API usability · Stateful signature schemes · Cryptographic agility

1 Introduction

1.1 The Need for Post-quantum Crypto Schemes

Quantum computers are the subject of ongoing research. With sufficient performance, they will be able to break the asymmetric schemes currently in use, such as RSA, DSA, ECDSA, and ECDH [10]. Their security is based on the prime factorization of large numbers and on the calculation of discrete logarithms,

© Springer Nature Switzerland AG 2019
N. Attrapadung and T. Yagi (Eds.): IWSEC 2019, LNCS 11689, pp. 221–240, 2019.
https://doi.org/10.1007/978-3-030-26834-3_13

respectively. For conventional computers, these are sufficiently difficult to solve if the parameters are suitably selected. However, this will no longer be the case when Shor's algorithms [28] are used on a sufficiently large quantum computer. The need for post-quantum cryptography (PQC), i.e. schemes secure enough to withstand a large quantum computer, is therefore evident.

In 2016, the National Institute of Standards and Technology (NIST) initiated a standardization process for PQC schemes.[1] These schemes are based on mathematical principles which are believed not to be vulnerable to quantum computer attacks. Hash-based schemes including XMSS [15], LMS [20] or SPHINCS [4] are possible candidates for post-quantum-secure algorithms. These schemes use hash functions to sign data and each signature requires a one-time key. Therefore, the private key contains a set of one-time keys. To record which keys have already been used, XMSS and LMS require a state. SPHINCS does not use a state and is therefore not considered within this work.

1.2 The Need for Usable Crypto APIs

The wrong usage of cryptographic functionality bears great risks and may lead to the leakage of personal data or identity theft. Therefore, the usage of cryptographic tools is indicated, if not for other reasons at least to be compliant with regulations such as GDPR[2]. This leads to an increasing integration of cryptographic functionality in software, including every-day applications such as instant messaging. Thus, more and more programmers, usually from other fields than cryptography, are using these APIs. Since these programmers are often unfamiliar with the required cryptographic principles, they struggle with the current APIs, which are too low-level for their needs [23]. Prior work shows that developers encounter problems using cryptographic APIs correctly [3,23,30]. Incorrect use of cryptographic APIs leads to insecure code, which in turn leads to an insecure application [11]. Therefore, easy-to-use APIs are playing an increasingly important role. Lazar et al. [19] have analyzed the vulnerabilities listed in the CVE database[3] and found that 83% of those vulnerabilities were caused by the incorrect use of cryptographic libraries, e.g. unsafe algorithms or hash functions were unknowingly applied, especially when using the default values provided by the API. Other errors included the use of weak random generators or private keys specified in the code.

Nadi et al. [23] have carried out 4 studies to point out typical problems regarding the use of cryptographic Java APIs. Problems included the time required to read resources such as the documentation, finding the correct sequence to call individual methods and insufficient knowledge about the domain. Similar to Acar et al. [3], the authors have identified good documentation, which should include examples for frequent tasks, as an important characteristic of a usable API. Many participants also mentioned the API design itself. The developers wish for

[1] https://csrc.nist.gov/Projects/Post-Quantum-Cryptography (2019-02-12).
[2] https://eugdpr.org (2019-06-07).
[3] https://cve.mitre.org (2019-03-09).

a *high-level* API that allows solving frequent tasks with only few method calls. Likewise, many developers requested tools to automatically detect faulty code and to provide code templates.

Stateful signature schemes introduce new challenges to the developer. In contrast to conventional signature schemes, the state of the private key changes with each signature. This property puts an extra burden to developers on their way to create secure applications. Even experienced developers may struggle with this new concept, as it differs from their experience and mental model. Our approach is to design an appropriate library that is easy to use from the developer's perspective. As to our knowledge, current implementations of stateful signature schemes (e.g. the XMSS implementation of Bouncy Castle[4]) do not automatically handle the states of the keys. How and whether the state is managed at all is entirely up to the developer, which indicates insufficient usability.

1.3 Goal and Approach

The goal of this paper is to design a cryptoagile, easy-to-use API for stateful and stateless signature schemes, focusing on a novel approach to handle the state of the private key.

To reach this goal, we investigate general design recommendations through literature research and conduct interviews with software developers (experts and non-experts) to collect a first set of requirements for our API. With these requirements, a prototype API for digital signatures, including the stateful scheme XMSS(MT), is designed and implemented. We evaluate our API in multiple iterations of small scale laboratory- and online-studies, improving our design with each iteration. These evaluation steps will provide us with an initial version of our API, ready to be used in a future, large-scale user-study.

The remainder of this work is structured as follows: we discuss related work (Sect. 2) as well as a new API layer for non-experts (Sect. 3), introduce our easy-to-use API for digital signatures, including stateful schemes (Sect. 4) and evaluate our API in user studies (Sect. 5). Section 6 concludes the paper and provides an outlook.

2 Related Work

Due to our interdisciplinary work, we discuss related work regarding stateful signature schemes and the management of their states (Sect. 2.1), the concept of cryptographic agility (Sect. 2.2), general recommendations for good API design (Sect. 2.3), research aiming to improve the usability of (cryptographic) APIs (Sect. 2.4) and methodology of online and laboratory studies (Sects. 2.5 and 2.6, respectively).

---

[4] https://www.bouncycastle.org/docs/docs1.5on/org/bouncycastle/pqc/jcajce/provider/xmss/package-frame.html (2019-02-21).

2.1 Stateful Signature Schemes and State Management

Buchmann et al. [8] describe the *eXtended Merkle Signature Scheme* (XMSS), an extension of the *Merkle Signature Scheme* (MSS) [22]. In XMSS, as well as in other hash-based signature schemes, the private key contains a limited set of one-time keys. As the name implies, these one-time keys can only be used once and therefore, only a limited amount of signatures can be performed with a single private key. The so-called state of the private key contains the information about which of the one-time keys have already been used. A further extension of XMSS is XMSSMT (*XMSS Multi Tree*). Through the introduction of subtrees in several layers, a practically unlimited number of signatures can be generated with comparable performance. Both schemes (XMSS and XMSSMT) are an Internet standard by the *IETF* [15].

Besides the reference implementation in C[5], a Java implementation for Bouncy Castle[6] exists as a lightweight API since version 1.57 and as a provider[7] implementation since version 1.58. Since we are using Java for our prototype implementation, we use the BC provider API as the basis to implement our own API (see Sect. 4). As of now, this is the only existing Java implementation for XMSS(MT).

McGrew et al. [21] investigate which measures must be taken to securely manage the states of hash-based signature schemes. They point out the danger of *cloning* in particular, especially by copying a virtual machine (VM) and with it the contained key material. A private key contained therein would then exist in two independent instances and could be used by both without synchronization.

Their work also mentions the risk of a *synchronization failure* in case the private key stored in the persistent storage fails to update at the same time or before the private key in RAM does, e.g. due to an application crash.

The authors consider the secure use of stateful signature methods possible in scenarios with dedicated hardware, but propose a hybrid scheme for general use. This includes scenarios that take the occurrence of *cloning* into account.

The paper also presents a strategy to increase efficiency by reserving states. The stateful private key is persistently stored n states in advance. Thus n keys are reserved for signing. Only if all reserved keys were used, the persistent storage would have to be accessed again.

We will take these considerations into account when designing our own API, by measures taken either in the implementation or in the documentation.

2.2 Cryptographic Agility

In order to respond to the constantly improving attacks on cryptographic schemes and primitives, APIs, cryptographic system components, and supporting libraries must be designed in a crypto-agile manner.

[5] https://github.com/joostrijneveld/xmss-reference (2019-03-09).
[6] https://www.bouncycastle.org (2019-03-09).
[7] https://docs.oracle.com/javase/8/docs/api/java/security/Provider.html (2019-03-09).

In RFC7696 [14] the agility of algorithms is described as follows: *"Algorithm agility is achieved when a protocol can easily migrate from one algorithm suite to another more desirable one, over time"*. RFC6421 [24] offers a similar definition.

Schneider described cryptographic agility in a more general way: *"Cryptographic agility refers to how easy it is to evolve or replace the hardware, software, or entire information technology (IT) systems being used to implement cryptographic algorithms or protocols (and, in particular, whether the resulting systems remain "interoperable")"* (opening remarks at a workshop on Cryptographic Agility and Interoperability [17]).

The replacement of algorithms is necessary, for example, if weaknesses are found in the algorithms or their implementation. But also simply the age of the algorithms and the increasing processing power of modern computers will make the use of more advanced algorithms a necessary step [14]. A faulty protocol design can also lead to weaknesses. However, in general, an simple exchange of cryptographic algorithms does not solve the problem. Instead, the protocol itself has to be adapted [14].

The crypto-agile integration of stateful schemes into IT systems poses a challenge. Compared to classical schemes, additional steps have to be performed in order to manage the state of the private key. Our goal is to take the above mentioned factors into account when designing our own API, providing a crypto-agile solution for stateful and stateless cryptographic schemes.

2.3 API Design

Several authors formulate usability principles for APIs or for systems in general [5,13,25]. We designed our API according to the following principles, as they best fit our use case and provide a good starting point: *easy to learn*, *easy to use*, *hard to misuse*, *safe default values*, *good documentation*, *easy to read and to maintain code*, *easily extensible*.

2.4 Usability of (cryptographic) APIs

Acar et al. [1] investigate the usability of cryptographic APIs in Python. Their work compares 5 of such APIs, 3 of which state to have good usability. The results show that simpler APIs usually lead to more secure code. Auxiliary functions, e.g. for storing the key material, should also be part of the API, although often this is not the case. Good official documentation was also considered as crucial. If no clear documentation is provided, the developer may turn to other sources (e.g. *StackOverflow*[8]), resulting in erroneous code [3].

Acar et al. [1] also introduced a new usability scale that better fits for the evaluation of APIs than the *System Usability Scale* (SUS, [6]). This new scale correlates with the *SUS*. 11 statements (see Appendix) about an API's usability are used to calculate a score between 0 and 100, where higher values represent a better usability. The statements have to be rated by the participants of a

[8] https://stackoverflow.com (2019-03-09).

usability study from *1 = strongly disagree* to *5 = strongly agree*. We make use of this usability scale during our small-scale user-studies.

Google is developing the cryptographic API *Tink*[9] with focus on usability. Accordingly, the API is *"secure, easy to use correctly, and hard(er) to misuse"*. *Tink* is the (unofficial) successor to *Keyczar*[10]. Currently, however, *Tink* does not support stateful signature schemes. The design and development of such APIs, as it is carried out in the work at hand, is therefore still necessary. To the best of our knowledge, scientific literature on Tink, especially on usability tests of the API, does not (yet) exist.

CogniCrypt is an extension for the Eclipse IDE [18]. It provides a wizard for secure code generation and static code analysis to continuously ensure the correctness of the code. While this helps the developer to create and maintain secure code, the usability of the cryptographic API itself is not improved. In contrast, our goal is to address the problem in an earlier phase by designing an easy-to-use cryptographic API, independent from any IDE or platform used by the developer. Tools like *CogniCrypt* may further be used complementary to further improve the process of code creation and maintenance.

2.5 Related Online Studies

Acar et al. [1] conducted their online study using a specifically developed online test environment which is described in detail by Stransky et al. [29]. The participants, most of whom were acquired from GitHub[11], were asked to solve a number of cryptographic problems using a randomly assigned API. After completing the tasks, they were asked to participate in an online survey. Gorski et al. [12] use a similar methodology, including test environment and participant acquisition, evaluating the integration of security warnings into the API.

For our own study, we are using the same test environment as mentioned above. While our methodology is similar, we focus on the usability of stateful signature schemes in particular. To our knowledge, this is the first work that examines the usability of such schemes.

2.6 Related Laboratory Studies

Scheller and Kühn [26,27] have conducted various laboratory studies to investigate factors that influence the usability of an API's methods and classes and to compare different configuration-based design concepts. For this purpose, participants were invited into a laboratory environment in which they were asked to solve a number of programming tasks. Screen recordings were made to analyze the results. These recordings made it possible to determine, for example, precise time values required to perform various steps, such as reading a specific section

[9] https://github.com/google/tink (2019-03-09).

[10] https://github.com/google/keyczar (2019-03-17).

[11] These were sent invitations by e-mail that had previously been extracted from git commits.

of the tutorial or documentation, or initializing a required class. In [27] there were three groups of 9 participants each. All groups were asked to solve a series of tasks with a different design concept. The time required for these tasks was analyzed and evaluated in combination with other collected information. In [26] a total of 20 participants took part in the study. They were divided into 2 groups of 10 participants each. For each group, a different API was provided to solve a number of tasks. Both studies were moderated by a supervisor sitting next to the participant during the entire execution period, giving explanations on each task.

As mentioned before, the online studies discussed in Sect. 2.5 served as an orientation as we use their methodology to evaluate our own API.

To summarize, we create and evaluate a crypto-agile, easy-to-use API design for digital signature schemes, including stateful ones, using various methods and principles described in this section.

3 A New Layer for Non-experts

There are established Crypto-APIs providing standardized access to cryptographic functionality. Prominent examples are the Microsoft Cryptography API: Next Generation[12] (CNG) for MS Windows applications and the Java Cryptography Architecture[13] (JCA) for Java-based applications. These are used by professional programmers to implement cryptography code in the respective language for the given platforms.

While these APIs provide very flexible access to cryptographic functionality, they also demand a detailed understanding of the underlying mechanisms. On the one hand, this allows the experienced developer to make detailed decisions on how to implement their IT security measurements; a possibility that is surely needed when out-of-the-box solutions do not fit. On the other hand, this also leaves much room for errors, especially when the programmer is not skilled in the use of cryptography; errors that should be avoided, especially when out-of-the-box solutions suffice.

In order to provide suitable cryptographic APIs for both, experts with the need for detailed tweaking and inexperienced programmers with everyday needs, our design consists of a new abstraction layer.

Figure 1 shows the conceptual integration of our new layer on top of the JCA. On the right hand side, the expert user directly accesses the JCA API as it comes with the Java Development Kit[14] (JDK). On the left hand side, the non-expert user employs an easy-to-use API that provides out-of-the-box cryptography methods for common cryptography tasks. The easy-to-use API in

[12] https://docs.microsoft.com/en-us/windows/desktop/seccng (2019-02-27).

[13] https://docs.oracle.com/en/java/javase/11/security/java-cryptography-architecture-jca-reference-guide.html (2019-02-21).

[14] https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html (2019-02-21).

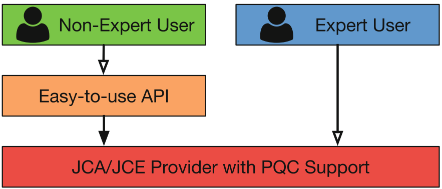

Fig. 1. Conceptual layer integration

turn, makes use of the JCA API. Note that the same design model can also be built upon other underlying standard APIs, e.g. CNG.

Besides providing a solution to the above described requirement of an expert API and to the demand for an error-proof API, this layered design comes with some additional advantages. Firstly, programmers that are already used to (and confident in) using the existing API (e.g. JCA or CNG) can continue to do so, they are not forced to use or learn an additional API. Secondly, the implementation of the easy-to-use API (by expert programmers) may be based (as in the example at hand) on an existing standard that already provides a suitable level of abstraction, especially regarding the exchange of underlying cryptographic algorithms. This way, our API inherits the cryptographic agility of that standard (see also Sect. 4.2).

4 EasySigner API

This section describes the *EasySigner* API[15], an easy-to-use API for digital signatures, providing a uniform interface for stateful and stateless signature schemes, both classical and post-quantum. A first prototype of this API, focusing on stateful signature schemes, is implemented in order to conduct small-scale usability tests (see Sect. 5). For the time being, we focus on a Java implementation of the API. We chose Java, as it is one of the most popular programming languages[16].

4.1 Requirements

Section 2 dealt with general design recommendations for (cryptographic) APIs. These were taken into account for the design of the *EasySigner* API.

Additionally, the results of four interviews with software developers from different German institutions were taken into account. These interviews were conducted to gain a deeper insight into the specific requirements for stateful signature schemes, also providing ideas and inspirations for the required methods, their naming and their placement. As is common in usability research, these

[15] Source code available at https://github.com/azeier-ucs/EasySigner-API.

[16] It was the most popular programming language in the StackOverflow developer survey 2018: https://insights.stackoverflow.com/survey/2018/ (2019-03-07).

interviews are intended to ensure that the API design meets the requirements and wishes of potential users.

Two of the interviewees were experienced Java developers with good knowledge of XMSS(MT) and cryptography in general (both rating their own Java (J) and crypto (C) knowledge with *high* or *very high*), working professionally with Java for 17 and 2 years, mostly on cryptographic tasks. The other two developers are less familiar with cryptography (C: *very low*, J: *medium*) or Java (J: *low*, C: *high*), respectively. All interviewees were acquaintances of the authors of this paper. The participation was voluntary and no incentive was given. These developers were chosen in order to gain insights from users with different skill sets. We will elaborate on the interviews in the following.

Interview Conduction. After familiarizing the interviewees with the topic at hand, they were asked about the challenges of using cryptographic APIs in general and stateful APIs in particular. In case of the two participants that were already familiar with the Bouncy Castle XMSS(MT) implementation, example code was presented at this point. For the other two participants, this was done at a later point during the interview. Here, the interviewees were asked to point out code fragments they felt were well or badly implemented. They were asked for example whether certain method calls appeared intuitive or whether the interviewees were unsure about their meaning. If an interviewee had already used the XMSS(MT) Java Provider, they could also report their own experiences.

Furthermore the interviewees were asked to write down their own ideas for an easy-to-use API for digital signatures, this way providing ideas for method names and required parameters as well as for the call sequence of related methods.

Interview Findings. According to all interviewees, a cryptographic API should be easy to use, even without any knowledge of cryptography or IT security, should provide secure default values that make it difficult to use the API incorrectly as well as a good documentation. In the following, several aspects are discussed in more detail:

Regarding the API's documentation, the interviewees stated missing examples for typical use cases. Instead, Google and StackOverflow are used, often resulting in insecure code, as already shown by Acar et al. [2].

In order to provide secure default values, *usage profiles* were suggested by some interviewees. Depending on the use case, e.g. for a Certification Authority (CA) or for code signatures, the respective usage profile contains *predefined values*[17] to be used by the developer. For some algorithms, suitable parameters already exist in literature. Hülsing et al. describe XMSS(MT) parameters for the use cases *Document and Code Signing* and *Communication Protocol* [16].

All interviewees preferred an automatic key management that does not require any interaction with the developer. This means the update of the key state is performed by the API with each signature. This requires the API to interact with the persistent storage of the key material.

[17] They are referred to as *predefined values* within the API's documentation, since the term profiles proved to be confusing in the first iteration of our usability tests.

Furthermore, backup strategies and parallel signing (which are also mentioned by Butin et al. [9]) were discussed. These aspects will not be considered further in this work, but will be part of future work.

Summarized Requirements. To summarize, Table 1 shows the requirements for the *EasySigner* API as determined through literature research and interviews. Additionally, the table states the source of each requirement and whether it was integrated in our prototype implementation (see Sect. 4.2).

Table 1. Requirements for the *EasySigner* API

	Requirements	Source	Prototype
Functional requirements	Usage profiles containing predefined values, e.g. for key generation	Interview	Yes
	Storing and loading key pairs from various storage formats, e.g. KeyStore file or HSM	Interview & Literature	Only KeyStore
	Automatic management of the key material, i.e. updating and persistent storage with every signature generation	Interview & Literature	Yes
	Reservation of states as described in Sect. 2.1	Literature	Yes
	Providing support for backups and parallel signing	Interview & Literature	No
Non-functional requirements	Easy to use for both experts and non-experts	Interview & Literature	Yes
	Good and complete documentation, including code examples	Interview & Literature	Yes
	In spite of the automated administration of the stateful key, the user should be aware that he is working with stateful keys and about the resulting risks	Interview & Literature	Yes

4.2 Design

We implemented a prototype of the *EasySigner* API. Since we are interested in the usability of stateful schemes in particular, only XMSS(MT) was implemented. We created dummy classes for RSA and ECDSA to demonstrate how stateless schemes fit within the API. This was also necessary to generate additional documentation and thus to be able to conduct a more realistic user-study.

We introduce a common abstraction layer for stateful and stateless signature schemes as presented in Sect. 3, meaning the state of a key will be handled by the API without any necessary actions from the developer. While other APIs, e.g. Java JCA, already provide ways to exchange the used algorithms in a modular way, we extend this ability to the exchange of stateless and stateful schemes. Therefore, the administration of the states must be within the scope of the API. Otherwise, additional method calls for stateful methods would be necessary. As our research show, this is also in the interest of the interviewed developers.

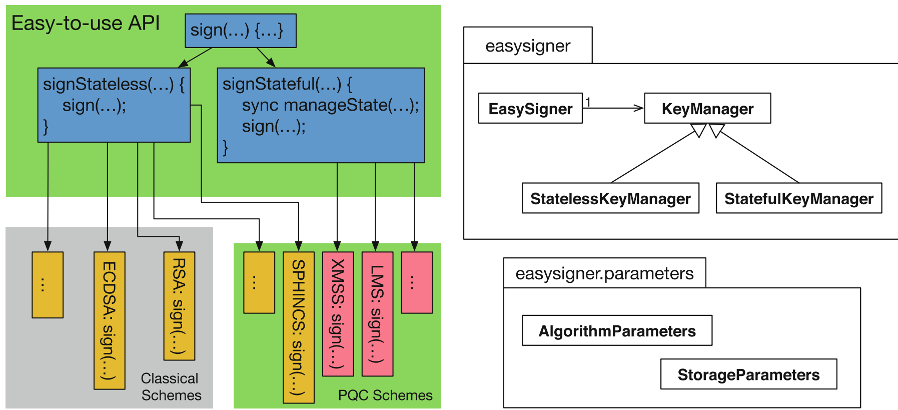

Fig. 2. API design

Fig. 3. API overview

For our implementation, we focus on non-expert users, trying to provide a high level API with the priority on easy usability. A JCA/JCE provider implementation, aiming at more experienced users, will be part of future work. Figure 2 shows a first design of the API. Depending on the signature scheme in use, the API calls the signing method for stateless or stateful schemes.

Figure 3 shows an overview of the *EasySigner* API. The API consists of the main class `EasySigner` that contains all methods needed for signing and verification. The class `KeyManager` is responsible for the management of the cryptographic keys. During initialization, the `EasySigner` object is given a `KeyManager` object, or has alternatively to be provided with the required parameters in order to create a `KeyManager` object by itself. There are two types of parameters: `AlgorithmParameters` and `StorageParameters`.

To give an example, the code to create a new XMSS key pair, stored in a `KeyStore`, looks like this:

```
AlgorithmParameters algorithmParameters = AlgorithmParameters.
    XMSSforSmallSignatures();
StorageParameters storageParameters = new KeystoreParameters("pathToFile",
    password);
EasySigner signer = EasySigner.withNewKeyPair(algorithmParameters,
    storageParameters);
```

This example uses *predefined values* for XMSS signatures. These default parameters ensure secure programming even for developers who lack the knowledge about which parameters to choose. These parameters may change over time, but can be renewed simply by updating the API or a corresponding configuration file. Changing any code is not necessary. This will be investigated further in future work. Nevertheless, by calling e.g.

```
AlgorithmParameters algorithmParameters = new XMSSParameters(20,
    XMSSParameters.SHA512);
```

the developer regains full control of the used parameters.

For the prototypical implementation of the API, the two *predefined values* `XMSSforSmallSignatures` and `XMSSMTforFastSigning` were taken from [16]. These two parameters are sufficient to test the concept of usage profiles and their placement within the API during the user study.

If a developer needs to use a different algorithm (e.g. XMSSMT), the respective line has to be changed to

```
AlgorithmParameters algorithmParameters = AlgorithmParameters.
    XMSSMTforFastSigning();
```

The rest of the code requires no changes. This also applies to subsequent operations, e.g. `verify` or `sign`, since the selection of the algorithm or the storage location is determined only once during initialization. A definition of the usage profiles as e.g. `String` values would make it possible to change the algorithm or parameters at runtime, without even changing a single line of code. This showed to be less usable in our study and we decided to employ the method demonstrated above. Further investigation of this (apparent) trade-off between cryptographic agility and usability will be part of our future large-scale study.

In case the `sign` method is called, either the `signStateless()` or `signStateful()` method will be executed depending on which algorithm is used (stateless or stateful). This is depicted in the architecture proposal in Fig. 2. To prevent corruption of the state, for example by multi-threading, `signStateful()` contains a *synchronized*[18] block for obtaining the current key as well as updating and storing the new key on the persistent storage. For this, the new updated key must first be stored before the old key is used. Otherwise a *synchronization failure* might occure (see Sect. 2.1).

The methods of the `KeyManager` class `createNewKeyPair()` and `loadKeyPair()` can either be called directly or by using the methods `withNewKeyPair()` and `withExistingKeyPair()` of the `EasySigner` class. These are *Factory* methods returning so-called *Singletons*. This prevents the initialization of several independent handles to the same Keypair. Once a new `KeyManager` is created with

[18] https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html (2019-03-13).

the same Keypair as an existing one, the method returns the earlier created object. The Keypair is identified by its path.

For the `StorageParameters` only `KeystoreParameters` were implemented in the context of this work. The use of other formats or HSMs to store the key material will not be discussed any further.

The storage location is indicated by the `StorageParameters`. These can be for example `KeystoreParameters`, in which case a Java `KeyStore` object is used to store the key material on the hard disk. The `KeystoreParameters` object therefore contains at least the path to the file and the password to the `KeyStore`. For other parameters, such as the aliases for public and private keys as well as for the certificate, predefined values are assumed. If the actual values differ, for example from an already existing key pair, another constructor, also allowing the specification of these parameters, may be used.

Depending on what `AlgorithmParameters` are passed to the `KeyManager` during initialization, the `KeyManager` automatically creates and returns an instance of the subclass `StatefulKeyManager` or `StatelessKeyManager`. The main difference between these subclasses is the management of the key material. The `StatefulKeyManager` ensures that the stateful key is updated and persistently stored again before each signature automatically. This prevents the user from making mistakes during the implementation of his application, which might result in the key material not being updated correctly. Furthermore, the same code can be executed regardless of the chosen scheme, without any adjustments for stateful methods. Exchanging the algorithm, in our case the `AlgorithmParameters`, is enough. This supports our goal of cryptographic agility.

For the realization of the state reservation strategy (see Sect. 2.1) the methods `signMultipleData()` in the `EasySigner` class and `updateKeyInAdvance()` in the `KeyManager` class were implemented. While `signMultipleData()` can be used for both stateful and stateless schemes, `updateKeyInAdvance()` is meaningful only with the use of stateful schemes, since otherwise no key updates are necessary. Nevertheless, for the sake of cryptographic agility, this method should also be available for stateless schemes, while calling it will have no effect.

5 User-Studies

To evaluate the usability of the designed API, a total of three iterations of small-scale user-studies were performed. Two in a laboratory setting with a total of 8 participants and one online study with 9 participants. After each iteration, the API was adjusted based on the results of the respective iteration. At the same time, we evaluated and adjusted the tasks, test environment and survey questions, leaving us prepared for our future large-scale study.

We conducted moderated laboratory studies. Since we had to change the location for almost each participant and therefore had to use a mobile laboratory setup, a non-moderated execution was not possible. The online study on the other hand was non-moderated, so we were able to gain results for both kinds of study. In the following, we will first present the procedures of the studies and then discuss the summarized results in a separate section.

Tasks. The user-study tasks were selected to test the complete functional range of the designed API. This includes the following API functions: Generating a key pair, storing and loading a key pair, creating a signature, verifying a signature, and increasing efficiency by reducing disk access.

Another crucial goal was to find out whether the participants were aware of working with stateful keys during the tasks.

For the first iteration, 4 tasks were given, each of them designed to cover at least one of the functions above. For the sake of a more realistic scenario and to save time, in the second iteration the same functions were tested within a single task. For the online study that same single task was used. Only the task's description was modified, adding details to better fit the online scenario and to compensate for the missing moderator.

Exit Survey. After completing the task(s), the participants were asked to answer questions about the API's usability. This was done to gain better insight into the difficulties that were encountered while solving the tasks and to receive further feedback. For the laboratory studies, this was done in form of an interview. In the online study, the participants were forwarded to a survey. Besides closed-ended questions, in which the participants could rate e.g. the correctness and security of their code and state whether they were aware of using a stateful scheme, they were also able to report usability issues in free text form.

In all iterations the participants were asked to rate a number of statements, leading to a usability score (see Sect. 2.4).

5.1 Laboratory Study

In the first two iterations of the evaluation, moderated usability tests were carried out with a total of 8 software developers. All developers had experience using Java, varying from only 2–3 years over 6–8 years to a maximum of 17 years. The experience with cryptography also varied from very high to almost non-existent. Before the study, none of the developers were familiar with XMSS(MT).

Furthermore, ethical considerations had to be taken into account. There were no ethical concerns regarding the laboratory study as the participation did cause no disadvantages of any kind. All participants took part in our study during their working hours with their employer's permission. No further risks were involved.

5.2 Online Study

After the completion of the laboratory studies, an online study was carried out to increase the number of participants and gain more significant data.

Setup. The online study was strongly based on the online studies conducted in [1,12]. From these studies, the test environment (*Developer Observatory*[19]),

[19] *Developer Observatory*, including setup guide, is available for download at https://github.com/developer-observatory/developer-observatory (2019-03-09).

including the consent form and introductory texts, was reused and only modified according to the deviating test subject.

Since *Developer Observatory* was originally implemented to test Python code, a few adjustments had to be made. In *Developer Observatory* the *Jupyter*[20] editor is used, which supports various programming languages via different kernels. For our online study the *SciJava*[21] kernel was chosen, because it worked most reliably during testing and allows the integration of custom JAR files.

Acquisition of Participants. Invitations to the online study were sent to mailing lists focusing on cryptography and online forums as well as posted to reddit boards about software, especially Java, development. As an incentive, three Amazon vouchers at a value of 100€ each were offered and randomly assigned to three participants after the study had been completed. Initially, there were no participants, presumably because of the required time to solve the task (about 1,5 hours was given as an estimate in the invitation) and the incentive not being guaranteed. Due to the lack of participants, additional invitations were sent to students and former students of our university. As a result, a total of 9 participants eventually completed the task (together with the exit survey). This is a sufficient amount for our pre-study, giving us a first insight into the current state of our API. The participants showed a broad variety regarding Java and cryptography knowledge, most of them being students or software developers.

5.3 Results

After each iteration the participants were asked to answer the API usability score in order to compare the results.

The result of the first iteration indicates a mediocre usability with an average of *68,06*. After integrating the feedback (e.g. about naming or method placement) into the API, a much better result could be achieved in the second iteration with an average of *87,08*. After the second iteration, only a small number of adjustments had to be made. In the following we will present the results of our online study in more detail. Since this was the latest study we conducted, all previous findings had been already integrated in the API's design, and therefore represents our current end result.

Table 2 (see Appendix) shows the determined API usability score of the *EasySigner* API. The table contains the 11 statements as described in Sect. 2.4. The score of *72,56* is slightly above the average value of *68* [7]. However, with a standard deviation of *12,86* there is also a strong dispersion of the results. In the following, the individual aspects of the score are discussed.

Comprehension. Statements 1–4 were rated mostly average or negative. These refer to the participant's comprehension of the API. Hence, the mere use of the API does not lead to a clear understanding of its functionality. The laboratory

[20] http://jupyter.org (2019-03-09).
[21] https://github.com/scijava/scijava-jupyter-kernel (2019-03-09).

studies showed that the documentation is hardly and rather reluctantly read, which may be a reason for the poor comprehension.

The lack of understanding of the API's functionality also leads to the fact that the participants were partly uncertain whether they had securely solved the task. They answered the question about the security of their solution with an average of *3,63/5* wherein half of the participants answered with a *3* or less.

Of the 9 participants, 6 did not realize they were working with a stateful signature scheme, even though it was mentioned in various places in the documentation and in a console output when generating or loading the key pair. This seems to confirm the assumption that most participants did not carefully read the documentation.

Documentation. The documentation was consistently perceived as helpful. With an average of *4*, a satisfactory result was achieved. Two faulty examples in the documentation were pointed out in the commentary by a participant, explaining his mediocre assessment of the documentation.

Also the *API Usability Score.* clearly shows that the documentation was perceived by the participants as very positive and helpful. They were able to find useful help easily (statement 7: *4,22*), helpful explanations (statement 8: *4,56*) and code examples (statement 9: *4,56*).

Naming and Usage. With an average of *4,89*, a nearly perfect score was achieved regarding naming. The usage of the API for solving the task was evaluated with an average of *4,11*. Analyzing the code written by the participants revealed that all of them had correctly and securely solved the task. As the survey showed, the participants themselves were confident about their solution.

Error Messages/Exceptions. Any error messages that occurred were also assessed as largely comprehensive by the participants (statement 10: *4,13*, statement 11: *4,5*, each with a minimum of *3*).

6 Conclusion and Outlook

In this paper, we present an easy-to-use API design for signature schemes, introducing a novel approach to handle stateful signature schemes such as XMSS(MT). The design is based on a literature review and findings from interviews, conducted with software developers (experts and non-experts). We evaluate our design through small-scale laboratory and online studies, using a prototypical Java implementation of our API. We achieved our goal as it was described in Sect. 1.3. We were able to achieve very good results regarding the documentation and the usage of the API (ratings of the respective statements of the *API Usability Score* with an average >4 out of 5). Among the participants of the online study, however, the use of the API only resulted in a mediocre understanding of its functions and the used algorithms (average ratings of the respective statements ≈ 3 out of 5). This also includes the developers' awareness

about working with stateful schemes. Most participants did not realize that they were using stateful keys. This may lead to security-critical errors. While the API ensures the update and persistent storage of the key material, it cannot prevent the key material from being duplicated outside the application or API. This could result in multiple use of a single state, ultimately compromising the key material. If the developer is not aware of this fact, he cannot assess whether or not such a scheme is suitable for a particular application.

Therefore, future work will investigate ways to make sure the developer fully aware of the statefulness of the schemes. This may include further improvement of the API's documentation or changing the API in a way that the statefulness is not transparent to the user, while still providing a crypto-agile solution.

Throughout the paper, various aspects for future work were mentioned. We summarize them here: (a) Investigating the possible trade-off between usability and cryptographic agility regarding the full fledged parameterization of the API, (b) Designing an update mechanism for the predefined values in our usage profiles, (c) Enhancing the API to support advanced functionality such as backing up the key material and perform parallel signing, (d) Transferring our key management approach to a JCA provider implementation.

The API will also be the subject of further improvements, including more usability tests at a larger scale. Part of these tests will be a comparison to other cryptographic APIs, e.g. regarding the time needed to solve certain tasks. It will be put into a larger context, being part of a comprehensive library for classical and post-quantum cryptography methods.

Acknowledgements. This project (HA proj. no. 633/18-56) is financed with funds of LOEWE – Landes-Offensive zur Entwicklung Wissenschaftlich-ökonomischer Exzellenz, Förderlinie 3 (State Offensive for the Development of Scientific and Economic Excellence). We thank our reviewers and the shepherd for their valuable feedback.

Appendix. API Usability Score of the Online Study

Table 2. API usability score of the online study.

Statement	O1	O2	O3	O4	O5	O6	O7	O8	O9	∅	σ
1: I had to understand how most of the assigned library works in order to complete the tasks	5	3	3	4	2	2	2	5	2	3,11	1,27
2: It would be easy and require only small changes to change parameters or configuration later without breaking my code	4	2	4	5	4	4	4	3	5	3,89	0,93
3: After doing these tasks, I think I have a good understanding of the assigned library overall	2	1	4	2	3	3	3	3	3	2,67	0,87

(*continued*)

Table 2. (*continued*)

Statement	O1	O2	O3	O4	O5	O6	O7	O8	O9	∅	σ
4: I only had to read a little of the documentation for the assigned library to understand the concepts that I needed for these tasks	3	2	5	2	5	2	5	2	3	3,22	1,39
5: The names of classes and methods in the assigned library corresponded well to the functions they provided	4	5	5	5	5	5	5	5	5	4,89	0,33
6: It was straightforward and easy to implement the given tasks using the assigned library	3	4	5	3	5	4	5	3	5	4,11	0,93
7: When I accessed the assigned library documentation, it was easy to find useful help	4	5	5	4	4	2	5	5	4	4,22	0,97
8: In the documentation, I found helpful explanations	4	5	5	4	4	4	5	5	5	4,56	0,53
9: In the documentation, I found helpful code examples	4	5	5	4	5	4	5	4	5	4,56	0,53
10: When I made a mistake, I got a meaningful error message/exception	4	3	5	0	4	4	4	5	4	4,13	0,64
11: Using the information from the error message/exception, it was easy to fix my mistake	4	3	5	0	5	5	4	5	5	4,50	0,76
Result	57,5	62,5	90	60,5	82,5	65	87,5	65	82,5	72,56	12,81

∅ = average
σ = standard deviation

References

1. Acar, Y., et al.: Comparing the usability of cryptographic APIs. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 154–171 (2017). https://doi.org/10.1109/SP.2017.52
2. Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M.L., Stransky, C.: You get where you're looking for: the impact of information sources on code security. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 289–305 (2016). https://doi.org/10.1109/SP.2016.25
3. Acar, Y., Stransky, C., Wermke, D., Weir, C., Mazurek, M.L., Fahl, S.: Developers need support, too: a survey of security advice for software developers. In: 2017 IEEE Cybersecurity Development (SecDev), pp. 22–26 (2017). https://doi.org/10.1109/SecDev.2017.17
4. Bernstein, D., et al.: SPHINCS: practical stateless hash-based signatures. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 368–397. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46800-5_15
5. Bloch, J.: Slides on 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. ACM (2006)
6. Brooke, J.: SUS - a quick and dirty usability scale. Usability Eval. Ind. **189**(194), 4–7 (1996)
7. Brooke, J.: SUS: retrospective. J. Usability Stud. **8**(2), 29–40 (2013)
8. Buchmann, J., Dahmen, E., Hülsing, A.: XMSS - a practical forward secure signature scheme based on minimal security assumptions. In: Yang, B.-Y. (ed.) PQCrypto 2011. LNCS, vol. 7071, pp. 117–129. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25405-5_8

9. Butin, D., Wälde, J., Buchmann, J.: Post-quantum authentication in OpenSSL with hash-based signatures. In: 2017 Tenth International Conference on Mobile Computing and Ubiquitous Network (ICMU), pp. 1–6. IEEE (2017). https://doi.org/10.23919/ICMU.2017.8330093
10. Chen, L., et al.: Report on Post-Quantum Cryptography. US Department of Commerce, National Institute of Standards and Technology (2016). https://doi.org/10.6028/NIST.IR.8105
11. 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. ACM (2012). https://doi.org/10.1145/2382196.2382205
12. 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. USENIX Association (2018)
13. Green, M., Smith, M.: Developers are not the enemy!: the need for usable security APIs. IEEE Secur. Priv. **14**(5), 40–46 (2016). https://doi.org/10.1109/MSP.2016.111
14. Housley, R.: Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms. BCP 201, RFC Editor (2015)
15. Hülsing, A., Butin, D., Gazdag, S., Rijneveld, J., Mohaisen, A.: XMSS: eXtended Merkle Signature Scheme. RFC 8391, RFC Editor, May 2018
16. Hülsing, A., Rausch, L., Buchmann, J.: Optimal parameters for $XMSS^{MT}$. In: Cuzzocrea, A., Kittl, C., Simos, D.E., Weippl, E., Xu, L. (eds.) CD-ARES 2013. LNCS, vol. 8128, pp. 194–208. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40588-4_14
17. Johnson, A.F., Millett, L.I. (eds.): Cryptographic Agility and Interoperability: Proceedings of a Workshop. The National Academies Press, Washington, DC (2017). https://doi.org/10.17226/24636
18. Krüger, S., et al.: CogniCrypt: supporting developers in using cryptography. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp. 931–936. IEEE Press (2017). https://doi.org/10.1109/ASE.2017.8115707
19. Lazar, D., Chen, H., Wang, X., Zeldovich, N.: Why does cryptographic software fail? A case study and open problems. In: Proceedings of 5th Asia-Pacific Workshop on Systems, pp. 1–7. ACM Press (2014). https://doi.org/10.1145/2637166.2637237
20. McGrew, D., Curcio, M., Fluhrer, S.: Leighton-Micali Hash-Based Signatures. RFC 8554, RFC Editor, April 2019
21. McGrew, D., Kampanakis, P., Fluhrer, S., Gazdag, S.-L., Butin, D., Buchmann, J.: State management for hash-based signatures. In: Chen, L., McGrew, D., Mitchell, C. (eds.) SSR 2016. LNCS, vol. 10074, pp. 244–260. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49100-4_11
22. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, New York (1990). https://doi.org/10.1007/0-387-34805-0_21
23. 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. ACM Press (2016). https://doi.org/10.1145/2884781.2884790
24. Nelson, D.: Crypto-Agility Requirements for Remote Authentication Dial-In User Service (RADIUS). RFC 6421, RFC Editor (2011)

25. Nielsen, J.: Usability Engineering. Elsevier, Amsterdam (1994)
26. Scheller, T., Kuhn, E.: Influencing factors on the usability of API classes and methods. In: 2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems, pp. 232–241 (2012). https://doi.org/10.1109/ECBS.2012.27
27. Scheller, T., Kühn, E.: Usability evaluation of configuration-based API design concepts. In: Holzinger, A., Ziefle, M., Hitz, M., Debevc, M. (eds.) SouthCHI 2013. LNCS, vol. 7946, pp. 54–73. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39062-3_4
28. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (1997). https://doi.org/10.1137/S0097539795293172
29. Stransky, C., et al.: Lessons learned from using an online platform to conduct large-scale, online controlled security experiments with software developers. In: 10th USENIX Workshop on Cyber Security Experimentation and Test, CSET 2017 (2017)
30. Xie, J., Lipford, H.R., Chu, B.: Why do programmers make security errors? In: 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 161–164 (2011). https://doi.org/10.1109/VLHCC.2011.6070393