Workgroup: cfrg
Internet-Draft:
draft-kaimindermann-securecryptoconfig-01
Published: 19 April 2021
Intended Status: Informational
Expires: 21 October 2021
Authors: K. Mindermann L. Teis
iC Consult GmbH

Secure Crypto Config

Abstract

Choosing secure cryptography algorithms and their corresponding parameters is difficult. Also, current cryptography APIs cannot change their default configuration which renders them inherently insecure. The Secure Crypto Config provides a method that allows cryptography libraries to change the default cryptography algorithms over time and at the same time stay compatible with previous cryptography operations. This is achieved by combining three things standardized by the Secure Crypto Config: (1) A process that is repeated every two years, where a new set of default configurations for standardized cryptography primitives is published in a specific format. (2) A Secure Crypto Config Interface that describes a common API to use cryptography primitives in software (3) using COSE to derive the parameters from output of cryptography primitives, otherwise future changes of the default configuration would change existing applications behavior.

Note to Readers

Comments are solicited and should be addressed to the GitHub repository issue tracker and/or the author(s)

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 21 October 2021.

Table of Contents

1. Introduction

1.1. Motivation

Cryptography needs standardization to work across various domains and to be interoperable between different implementations. One domain that is not covered sufficiently by cryptography standards is the selection and maintenance of cryptography algorithms and their

parameters. Choosing an appropriate and secure cryptography algorithm alone is difficult. Yet, even more difficult is the choice of the required and matching parameters for that algorithm (e.g. Argon2 has 10 input parameters). After the choice has been made, all involved parties need to use exactly this configuration. There is no specification on how the chosen cryptography configuration should be stored, distributed and retrieved. Furthermore, supporting more than one configuration or being able to add future configurations is not defined. That reduces software compatibility and increases maintenance efforts.

Cryptography algorithm implementations, regardless of for one algorithm or multiple ones, offer some kind of Application Programming Interface for developers to use the algorithms. Yet, in many cases these interfaces provide no abstraction from the underlying algorithm but expose much of the internal states and parameters. Also the more abstracting interfaces, usually found in the standard libraries of programming languages, require users to have much cryptography experience to use them correctly and securely. Moreover, even approaches that tried to increase usability by providing defaults, these defaults become quickly outdated but cannot be changed in the interface anymore as applications using these defaults rely on that functionality.

It sounds a lot like a problem for software engineering and not for cryptography standardization. But software engineering alone cannot provide a programming interface for cryptography algorithms that also works for future algorithms and parameters and at the same time is able to change the default implementation easily. Both the choice of the algorithm/parameters and the default behavior must be automated and standardized to remove this burden from developers and to make cryptography work by default and in the intended secure way.

The Secure Crypto Config approaches this problem first by providing a regularly updated list of secure cryptography algorithms and corresponding parameters for common cryptography use cases. Second, it provides a standardized Application Programming Interface which provides the Secure Crypto Config in a misuse resistant way to developers. Third, it leverages an already standardized format ([RFC8152]) to store the used parameters alongside the results of cryptography operations. That ensures that future implementations can change their default cryptography algorithms but still parse the used configuration from existing data and perform the required cryptography operations on it.

Each of these approaches could be used on its own. Yet, the combination of them allows software to be easier to maintain, more compatible with other cryptography implementations and to future security developments, and most importantly more secure.

The Secure Crypto Config makes common assumptions that are not true for all possible scenarios. In cases where security experts are indeed involved and more implementation choices have to be made, the Secure Crypto Config still allows the usage of predefined or even custom cryptography algorithms and parameters.

1.2. Terminology

1.2.1. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2.2. Terms

The terms "API", "attack", "authenticity", "availability", "break", "brute force", "cipher", "ciphertext", "classified", "classification level", "confidentiality", "cryptographic hash", "encrypt", "encryption", "initialization vector (IV)", "integrity", "key", "mode", "nonce", "password", "plain text", "plaintext", "protocol", "security", "security level", "threat", "trust", in this document are to be interpreted as described in [RFC4949].

The term "hash" is used as a synonym for "cryptographic hash".

The term "cryptographic primitive" is used in this document to describe a generic building block used in the field of cryptography e.g. Hashing, symmetric encryption.

1.3. Use Cases

1.3.1. Secure Crypto Config Use Cases

The Secure Crypto Config has the following main use cases:

* Centralized and regularly updated single source of truth for secure algorithms and their parameters for most common cryptography primitives and use cases.

* Both machine and human readable format to specify the above mentioned cryptography algorithm/parameter configuration. The format is also extensible to allow others (e.g. governmental or commercial organizations) to define their own set of cryptography configurations.

* Standardized cryptography API that not uses the Secure Crypto Config for the selection of the most recent cryptography

configurations but also uses a standardized cryptography operation output format to enclose the chosen parameters.

1.3.2. Cryptography Use Cases

The Secure Crypto Config covers cryptography algorithm and parameter configurations for widely used cryptography use cases defined in the following sections.

1.3.2.1. Symmetric Encryption

Symmetric Encryption is an important cryptographic primitive especially as it is usually multiple magnitudes faster both for encryption and decryption than asymmetric cryptography. Yet, the secret has to be shared with all participants.

The only expected input parameters by cryptography users:

- *plaintext
- *secret key

Expected output: ciphertext.

Additional Parameters often required in practice:

- *Algorithm
- *Block-Mode
- *IV
- *Padding-Mode
- *Authentication Tag size

Possible secure usage:

- *A256GCM;3;AES-GCM mode w/ 256-bit key, 128-bit tag
- *ChaCha20/Poly1305;24;ChaCha20/Poly1305 w/ 256-bit key, 128-bit tag

1.3.2.2. Asymmetric Encryption

Besides symmetric encryption is asymmetric encryption another important cryptographic primitive to considered.

The only expected input parameters for encryption:

- *plaintext

*public key

Expected output: ciphertext.

The only expected input parameters for decryption:

*ciphertext

*private key

Possible secure usage:

*RSAES-OAEP w/ SHA-512,-42,RSAES-OAEP w/ SHA-512

1.3.2.3. Hashing

Hashing is an important cryptographic primitive and is often needed as a part of many other cryptographic use cases e.g. password derivation.

The only expected input parameters by cryptography users:

*plaintext

Expected output: hash.

Possible secure usage:

*SHA-512 (TEMPORARY - registered 2019-08-13, expires 2020-08-13);-44;SHA-2 512-bit Hash

*SHAKE256 (TEMPORARY - registered 2019-08-13, expires 2020-08-13);-45;256-bit SHAKE

1.3.2.4. Password Hashing

The secure storage of passwords requires hashing. Yet, password hashing requires that the hashing can not be performed very fast to prevent attackers from guessing/brute-forcing passwords from leaks or against the live system. E.g. it is totally fine for users if the login takes 0.1 seconds instead of microseconds. This results in special families of hash algorithms that offer additional tuning parameters.

The only expected input parameters by cryptography users:

*plaintext

*hash-algorithm

Expected output: hash.

Possible secure usage:

- *Argon2id

1.3.2.5. Key Generation

- *[ ] TODO should key generation be considered? (Symmetric/ Asymmetric)

A key is necessary for many cryptographic use cases e.g. symmetric and asymmetric encryption. Therefore, key generation is an important part while implementing cryptographic code.

The only expected input is the intended use case.

Expected output: key.

Possible secure generation:

- *Use of CSPRNG

- *Keys derived via derivation function from passwords/other keys

1.3.2.6. Digital Signatures

Signing is an important and often needed cryptographic use case. It is based on the principle of asymmetrical encryption.

The only expected input parameters for signing:

- *plaintext

- *private key

Expected output: signature.

The only expected input parameters for verifying the signature:

- *signature

- *public key

Expected output: valid/not-valid.

Possible secure usage:

- *ECDSA

- *ES512;-36;ECDSA w/ SHA-512

2. Requirements and Assumptions

2.1. Requirements

In the following, all requirements are listed that regard the Secure Crypto Config or the Secure Crypto Config Interface.

- *Security Level Requirements: The Secure Crypto Config should define different security levels. E.g. information has different classification levels and longevity. Additionally, cryptography operations could not or much slower perform on constrained devices, which should also be handled with the security levels. For each security level, the consensus finding process and entities shall publish a distinct secure crypto config.

- *Consensus Finding Process and entities:

  - -The Secure Crypto Config must be renewed regularly.

  - -The Secure Crypto Config must be renewable on-demand.

  - -There must be a guideline on which entities must agree to publish a new Secure Crypto Config.

  - -There must be a guideline on which entities may participate in the consensus finding process and how they may do so.

  - -There must be a guideline on how to determine broad availability of both cryptography algorithms and chosen parameters.

- *Publication Format and Distribution Requirements:

  - -General:

    - oThe Secure Crypto Config must be easily publishable by the committee.

    - oStandardized unique and distinct names for (1) cryptography algorithms (2) their parameters and (3) the combination of the algorithm with set parameters. Otherwise, ambiguity would make it harder for developers and cryptography implementors to make correct and secure choices.

    - oThere must be a versioning that allows to distinguish between Secure Crypto Configurations and what is the latest Secure Crypto Config.

    - oThere must be a deprecation process that ensures usage of outdated/insecure Crypto Configurations cases.

      - There must be an official source where this Secure Crypto Config is maintained and can be obtained from (e.g. via the WWW).

      - The official source format of the Secure Crypto Config must be cryptographically protected to ensure its integrity and authenticity.

      - Other published formats derived from the source format (e.g. for human readability on a webpage) do not have to be cryptographically protected but should be generated automatically from the source format.

      - The official source should also provide information about the Secure Crypto Config Interface that should be utilized for the application of the Secure Crypto Config.

      - The Secure Crypto Config must specify how it can be extended (e.g. more security levels) and how derivatives work.

   - Human readable

      - The Secure Crypto Config must have a human readable format.

      - The Secure Crypto Config must allow non-experts to find secure cryptography algorithms and appropriate parameters for common cryptography use cases.

      - The Secure Crypto Config human readable publication format should only use easy to comprehend data structures like two-dimensional tables.

   - Machine readable

      - Cryptography libraries, regardless of the programming language, should be able to directly map (without extensive parsing) the Secure Crypto Config to their implementation

      - Must be easy to verify which Secure Crypto Config is used / was used (e.g. in Continuous Integration platforms)

      - Must be easy to verify the authenticity of the Secure Crypto Config (e.g. is this really what the CFRG has published?)

* Cryptography library integration requirements:

   - Easy to integrate by cryptography libraries

- Experts should still be able to use/access the unaltered output of cryptographic primitives

- Recommendation what should be the default Secure Crypto Config for a cryptography library (e.g. should it be the one with the highest security level or *only* a weaker one?)

- Recommendation of what should a cryptography library do if it can not support the parameters specified in the latest Secure Crypto Config. (E.g. key size for RSA would be n*2 and the library supports only n)

- Recommendation on how a cryptography library should integrate the Secure Crypto Config so that it is up to date as soon as possible after a new Secure Crypto Config has been published

* General Requirements:

- Interoperability with other standards/formats (e.g. [RFC8152])

- The Secure Crypto Config should cover most common cryptography primitives and their currently broadly available and secure algorithms.

- The Secure Crypto Config should be protected against attackers as defined in Section 3.6.5

- The Secure Crypto Config should prevent non-experts to configure cryptography primitives in an insecure way.

- The Secure Crypto Config should not prevent experts from using or changing all parameters of cryptography primitives provided by a cryptography library/API.

2.2. Assumptions

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

3. Security Levels

The Secure Crypto Config must be able to provide a secure parameter set for different security levels. These security levels depend on the following security constraints: **Information classification (Secret, Confidential), Longevity (less than one day, more than a day), Constrained devices (constrained, not constrained)**. They are defined in Section 3.6 below. The Secure Crypto Config provides 5 common security levels for which official algorithm/parameter choices are published.

3.1. Security Level 1 - Low

Confidential information, regardless of the other two constraints

3.2. Security Level 2

Secret information, less than one day longevity, constrained device

3.3. Security Level 3

Secret information, less than one day longevity, non-constrained device

3.4. Security Level 4

Secret information, more than a day longevity, constrained device

3.5. Security Level 5 - High

Secret information, more than a day longevity, non-constrained device

3.6. Security Level Constraints

3.6.1. Information classification

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

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

3.6.2. Longevity

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

3.6.3. Constrained Devices

For cryptography often complex computations have to be executed. Yet, not all environments have the same hardware resources available. E.g. it is not always the case that the used processors have dedicated cryptography hardware or even specialized execution units or instruction sets like [AESNI]. Detailed discussion and definitions can be found in [RFC7228]. Yet, their definitions are too concrete to be used in the Secure Crypto Config's standardized security levels. Therefore, the Secure Crypto Config defines constraint devices not based on concrete processing power (e.g. 100k instructions per second):

A device is constrained when it has **multiple orders of magnitudes** fewer resources than a current (not a new one, but broadly in use at the time of publication of a Secure Crypto Config!) standard personal computer.

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

3.6.4. n-Bit-Security

n-Bit Security Level:

A cryptographic mechanism achieves a security level of n bits if costs which are equivalent to 2^n calculations of the encryption function of an efficient block cipher (e.g. AES) are tied to each attack against the mechanism which breaks the security objective of the mechanism with a high probability of success. BSI

3.6.5. Attacker Resources and Capabilities

The Secure Crypto Config considers only the following same attacker resources and capabilities for all standardized security levels:

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

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

4. Consensus Finding Process and entities

To provide a Secure Crypto Config, it is necessary to agree upon a secure and appropriate cryptographic parameter set for each security level (see Section 3). This must happen in a common consensus finding process which takes place during regular intervals. The consensus finding process is based on the established RFC process during which the Secure Crypto Config Working Group decides in cooperation with the Crypto Forum Research Group (CFRG) and other institutions like the Bundesamt fuer Sicherheit in der Informationstechnik (BSI) or the National Institute of Standards and Technology (NIST) for a set of secure parameters. After the successful decision, the agreed on parameters can be added in the defined publication data structures (see Section 5.4) and provided on the repository platform.

4.1. Consensus Finding

Consensus must be found two years after the last consensus was found. This ensures that there is a new Secure Crypto Config every

two years, even if the configuration itself has not changed. There is a regular process and an emergency process to release Secure Crypto Configurations.

4.1.1. Regular Process

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

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

* (2) Six months **Consensus finding phase** during which all participating entities must agree on a common Secure Crypto Config.

* (3) Six months **Publication phase** ensures the publication of the final Secure Crypto Config AND allows the Secure Crypto Config Interface and other cryptography implementations to integrate the necessary changes.

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

4.1.2. Emergency Process

* [ ] TODO How can the Working Group alter the Secure Crypto Config IANA registry / or use the RFC Errata process?

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

1. Determine the insecure configuration.

2. Remove the insecure configuration.

3. Publish the revised Secure Crypto Config with a new patch version.

4. Mark the old (unrevised) Secure Crypto Config as deprecated.

Examples for emergency cases are drastically better brute force algorithms or brute force performance (e.g. quantum computers/ algorithms), drastically discovered flaws in proposed algorithms and their configurations.

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

4.1.3. Requirements for Selection of Cryptography Algorithm and Parameters

The Secure Crypto Config MUST only propose cryptography algorithms and parameters that fulfill the following requirements:

* Cryptography algorithms and parameters have stable implementations in at least two different programming languages.

* Cryptography algorithms and parameters have a defined standard to store the algorithm and parameter identification alongside the result (e.g. like [RFC8152]). This is required to ensure cryptography operation results can be processed even if the default parameters have been changed or the information has been processed with a previous version of the Secure Crypto Config.

* Cryptography algorithms that support parametrization to adapt to increased brute force performance and to allow longevity of the algorithm especially for hardware optimized implementations.

The Secure Crypto Config SHOULD only propose cryptography algorithms and parameters that fulfill the following requirements:

* Cryptography algorithms and parameters are defined in a globally accepted standard which was suspect to a standardization process.

* Cryptography algorithms and parameters are licensed under a license that allows free distribution.

4.2. Entities

Entities that participate in the proposal phase SHOULD have significant cryptography expertise. Entities that participate in the consensus finding phase MUST have significant cryptography

expertise. Cryptographic expertise is defined by the Secure Crypto Config Working Group or the CFRG.

5. Publication Format and Distribution

In general the Secure Crypto Config is published via JSON [RFC8259] files in an official repository. The Secure Crypto Config also utilizes IANA registries, see {#IANA}.

5.1. Versioning

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

5.2. Naming

The naming of official released SCCs must follow this format:

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

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

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

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

5.3. Secure Crypto Config IANA Registry

NOT NEEDED?, as the Secure Crypto Config uses other registries, e.g. COSE. No final decision, yet.

*[ ] TODO Naming convention. Specification depending on crypto use case?

*[ ] TODO dash character "-" not possible in enum!

The Secure Crypto Config requires one IANA Registry with the following columns:

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

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

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

5.3.1. Example for Secure Crypto Config IANA Registry

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

Table 1

5.3.2. Utilized Algorithm Registries

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

* CBOR Object Signing and Encryption (COSE)
* AEAD Algorithms
* Named Information Hash Registry

Used registries must define all required parameters for an algorithm to implement it without ambiguity. E.g. implementations must not be

able to choose other parameter values for these predefined cryptography algorithm and parameter combinations.

5.4. Data Structures

For each defined security level, a distinct JSON file must be provided. These files must adhere to the common schema and shown in Figure 1 and described in the following.

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

Figure 1: General JSON format

*SecurityLevel: Contains the number of the corresponding Security Level of the Secure Crypto Config

*PolicyName: Contains the name of the corresponding Secure Crypto Config according to the naming schema defined in Section 5.2

*Publisher: Contains an array of all parties that participated in the consensus finding process

  -name: Name of the participating party

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

*Version: Contains version in the format defined in Section 5.1

*PolicyIssueDate: Date at which the Secure Crypto Config was published in the format: YYYY-MM-DD

*Expiry: Date at which the Secure Crypto Config expires in the format: YYYY-MM-DD

*Usage: Contains an array of objects for each cryptographic use case defined in Section 1.3.2.

  -For each cryptographic use case, at least two agreed upon algorithms (see Section 4) with necessary parameters are included. Each of these algorithms with its parameters is specified with its unique identification name defined in a IANA registry used by the Secure Crypto Config.

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

5.5. Human readable format

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

5.6. Official Secure Crypto Config Repository

5.6.1. Location of Secure Crypto Config Repository

The needed Secure Crypto Config files should be published at an official GitHub repository. There, all current versions will be provided during the interval of the Publication phase (see Section 4.1.1). Additionally, all previously published files are still stored at this location even if new versions are published.

5.6.2. Format of Secure Crypto Config Repository

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

Figure 2: Example for Secure Crypto Config Repository content

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

Figure 3: Example for Secure Crypto Config Repository content with custom naming scheme

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

5.6.3. Integrity/Signing process

*[ ] TODO what kind of signing process should be used?

-[ ] GPG?

-[ ] openssl?

-[ ] Git GPG signed commits?

-[ ] Use an SCC recommended signing algorithm/format

*[ ] Can two signatures be put in one signature file? Should they be put in the same file?

*[ ] Public Key distribution?! (In GitHub repo?)

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

6. Secure Crypto Config Application Programming Interface (API)

This section describes the application programming interface (API) that provides the Secure Crypto Config. The Secure Crypto Config Interface is generic and describes the API that should be used by each programming language.

6.1. Semantic Versioning

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

6.2. Deployment of (custom) Secure Crypto Config with Interface

There are two different possibilities to work with the Secure Crypto Config: - The preferred option is to use the Secure Crypto Configs that will be delivered within the Interface. In each new Interface version the current Secure Crypto Configs will be added such that always the latest Secure Crypto Configs at the time of the Interface release will be supported. Previous Secure Crypto Configs will remain inside the Interface such that also older ones can still be used. - Another option is to define a specific path to your own/ derived versions of the Secure Crypto Configs with the same structure of the files as described in Section 5.4 but with other values than in the official ones.

The Interface will process the Secure Crypto Configs as follows:

1. Check if the path to the Secure Crypto Configs is a valid one.

2. Check if the configs folder exists.

3. For each folder following configs in the hierarchy look inside that folder and check the existence of JSON files that need to be considered. This check will happen recursively for all folders inside the hierarchy.

4. For every JSON file found, look if there exists a signature. If one is given, check if the signature is valid for the corresponding file.

5. Every file with a valid signature will be parsed and further processed.

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

1. Read out all information of all JSON files that need to be considered. The information of each file is stored in a corresponding object. With this procedure all JSON files need to be read only once which will contribute to the performance.

2. Parsing of security level: Check if it is a positive integer. All files not containing an (positive) integer number as security level value will be discarded.

3. Parsing of algorithm identifiers: Only the algorithm identifiers that are supported by the Interface will be considered and stored inside the corresponding object. The supported algorithms are specified inside the interface (e.g. with an enmum).

4. Parsing of the version of all files: All files with values in the wrong format (see Section 5.1) will be excluded from further processing. Find the latest (according to version) Secure Crypto Config file with the highest appearing security level (determined in previous step). The path to this file will be used as default path used for each cryptographic use case if nothing else is specified by the user. If two or more files with identical levels and version number are found, only the first one will be used, others are discarded.

5. The unique algorithm identifiers for the execution of a specific cryptographic use case will be fetched from the corresponding object (representing the JSON file determined beforehand) at the time the users invoke a method for a specific cryptographic use case. The Interface will also provide a possibility to choose a specific algorithm (out of the supported ones) for executing the desired use case. In this case the specified algorithm is used. The identifiers will be compared with the supported ones in order of their occurrence inside the file. If one matching identifier is found it will be used for execution. If it is not a matching one the value will

be skipped and the next one will be compared. If none of the algorithms inside the selected Secure Crypto Config can be found an error will occur.

6.2.1. Delivery of Secure Crypto Config with Interface

Each Secure Crypto Config Interface must be published in such a way that it uses (a copy of) the recent Secure Crypto Config repository.

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

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

6.2.2. Using a custom Secure Crypto Config Repository

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

6.2.3. Integrity Check

*[ ] TODO which public keys should be used? (See above Integrity/ Signing process Public Key distribution?!)

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

6.2.4. Methods and Parameters

Intended methods and parameters included in the Java interface are described in Figure 5.

6.2.4.1. Supported Algorithm Parameter Types

- [ ] TODO What is with parameters that have to be chosen during runtime? (e.g. the length of the nonce can be specified but not its content?) Maybe refer to how the PHC String Format describes how parameters must be defined and only allow constants and csprng generated content?

Cryptography algorithms require different parameters. The Secure Crypto Config Interface considers the following types of parameters:

- Parameter Size (e.g. key length in bit)
- Parameter Counter Content (e.g. nonce)
- Parameter Secure Random Content (e.g. nonce)
- Parameter User Automatic Tunable Content (e.g. memory consumption for Argon2 password hashing algorithm)
- Parameter User Defined Content (e.g. plaintext and key for symmetric encryption)
- Parameter Compound Parameter Content (e.g. counter + random = nonce)

6.2.5. Automatic Parameter Tuning

- [ ] TODO is it possible to define new algorithm/parameter combinations on the fly (in extensions/derivations) or are only SCC IANA registry identifiers allowed/usable?

It should be possible to have user specified parameters such as the key/nonce length explicitly given by the user, but also a performance mode that evaluates for each configuration and gives back a prioritized list for each configuration. In this way, it is possible to select parameters depending on systems properties. Such a parameter choice would be beneficial e.g. in the case of Argon2 in which one parameter for the memory usage must be given. This choice should be chosen based on the corresponding system. That kind of parameter selection can be seen e.g. in Libpasta Tuning, which returns a secure parameter set depending on executed evaluations.

6.2.6. Output of readable Secure Crypto Config

A Secure Crypto Config Interface must offer the following additional methods regarding the configuration - A method that returns a human readable version of the currently used Secure Crypto Config - A method that returns the currently used cryptography algorithm and parameters for a given use case - A method that validates the

content of a Secure Crypto Config JSON file and one or more signatures

6.3. TODOs

* The SCC could be provided on a suitable platform (?) and is accessible over the network (adversaries? e.g. http connection)
  - [ ] e.g. should there be constants like "SCC_TOP_SECRET_LATEST" and "SCC_TOP_SECRET_LATEST".
  - [ ] And like "SCC_TOP_SECRET_LATEST.AES" which points always to the latest Secure Crypto Config definition for AES parameters.
* [ ] TODO how should cryptography implementations, that implement/support SCC, generate the parameters?
* [x] What kind of parameters can be chosen based on the Secure Crypto Config? => E.g. Should be all except the plaintext and the key for encryption algorithms. Also, many parameters can be generated based on cryptographically secure random numbers.
* [x] TODO The Secure Crypto Config Interface should include a performance evaluation mode which evaluates the performance of each configuration and returns a prioritized list for each configuration. E.g. cf. Libpasta Tuning

7. Cryptography Library Implementation Specification

Cryptography libraries should provide the above mentioned Secure Crypto Config Interface. Until a common cryptography library provides the Secure Crypto Config Interface itself, there should be wrapper implementations that provide the Secure Crypto Config Interface and make use of the programming languages' standard cryptography library.

8. Cryptography Algorithm Standards Recommendation

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

9. Security Considerations

* [x] TODO are some of the listed common issues relevant?: TypicalSECAreaIssues

* [x] TODO check if security considerations of TLS 1.2 are relevant, especially appendix D, E and F

* [ ] TODO Are these appropriate security considerations?

9.1. Consensus Finding

* Only trustworthy and cryptographic specialized entities should participate in the publication process of the Secure Crypto Config. Otherwise a Secure Crypto Config with a weak and insecure parameter set could be provided.

9.2. Publication Format

* The operators of the Secure Crypto Config must ensure that potential unauthorized parties are not able to manipulate the parameters of the published Secure Crypto Config. Countermeasures to this are in place by utilizing git's gpg signatures and integrity as well as signatures for the published Secure Crypto Config files as well.

9.3. Cryptography library implementation

* Integrity must be ensured if potential users want to fetch the provided Secure Crypto Config from the corresponding platform over the network e.g. by using a signatures.

* Users should only trust Secure Crypto Config issued from the original publisher with the associated signature. Users are responsible to verify the provided signatures.

9.4. General Security Considerations

9.4.1. Special Use Cases and (Non-)Security Experts

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

9.5. Security of Cryptography primitives and implementations

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

9.5.1. Security Guarantees

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

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

9.5.2. Threat Model / Adversaries

There are different possibilities in which a potential adversary could intervene during the creation as well as after the publication of the Secure Crypto Config. These attack scenarios must be considered and prevented.

* **Process:** During the creation process, it is necessary for selected institutions to agree on a secure parameter set. It could be possible that one party wants to influence this process in a bad way. As a result, it could be agreed on weaker parameter sets than originally intended.

* **Publication:** After the publication of the Secure Crypto Config a potential attacker could gain access to the provided files on the corresponding platform and change the content to an insecure parameter set.

* **Content:** Depending on the distribution method of the Secure Crypto Config, it is also possible that an attacker could change the content of the Secure Crypto Config as man-in-the-middle. Especially if an http connection is used to obtain the Secure Crypto Config, this will be a serious problem.

10. IANA Considerations

* [] TODO Are there IANA Considerations?

* [] TODO May add reference to own registry

The data structure (see Section 5.4) defined in this document uses the JSON format as defined in [RFC8259].

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <https://www.rfc-editor.org/rfc/rfc4949>.

[RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <https://www.rfc-editor.org/rfc/rfc8152>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/rfc/rfc8259>.

11.2. Informative References

[AESNI] Gueron, S., "Intel Advanced Encryption Standard (AES) Instruction Set White Paper", 2010, <https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf>.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, DOI 10.17487/RFC2743, January 2000, <https://www.rfc-editor.org/rfc/rfc2743>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/rfc/rfc5116>.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <https://www.rfc-editor.org/rfc/rfc5652>.

[RFC5698] Kunz, T., Okunick, S., and U. Pordesch, "Data Structure for the Security Suitability of Cryptographic Algorithms

(DSSC)", RFC 5698, DOI 10.17487/RFC5698, November 2009, <https://www.rfc-editor.org/rfc/rfc5698>.

[RFC6916] Gagliano, R., Kent, S., and S. Turner, "Algorithm Agility Procedure for the Resource Public Key Infrastructure (RPKI)", BCP 182, RFC 6916, DOI 10.17487/RFC6916, April 2013, <https://www.rfc-editor.org/rfc/rfc6916>.

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <https://www.rfc-editor.org/rfc/rfc7228>.

[RFC7696] Housley, R., "Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms", BCP 201, RFC 7696, DOI 10.17487/RFC7696, November 2015, <https://www.rfc-editor.org/rfc/rfc7696>.

Appendix A. Examples

A.1. JSON Secure Crypto Config

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

Figure 4: Example for JSON format

Appendix B. Example Java Interface using Secure Crypto Config

```
package org.securecryptoconfig;

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

import COSE.CoseException;

public abstract interface SecureCryptoConfigInterface {

//Symmetric
public AbstractSCCCiphertext encryptSymmetric(AbstractSCCKey key, Plaint
    throws SCCException;

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

public AbstractSCCCiphertext reEncryptSymmetric(AbstractSCCKey key, Abst
    throws SCCException;

public PlaintextContainerInterface decryptSymmetric(AbstractSCCKey key,
    throws SCCException;

// Asymmetric
public AbstractSCCCiphertext encryptAsymmetric(AbstractSCCKey key, Plain
    throws SCCException;

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

public AbstractSCCCiphertext reEncryptAsymmetric(AbstractSCCKey key, Abs
    throws SCCException;

public PlaintextContainerInterface decryptAsymmetric(AbstractSCCKey key,
    throws SCCException;

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

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

public AbstractSCCHash updateHash(PlaintextContainerInterface plaintext,
    throws SCCException;

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

```
public boolean validateHash(PlaintextContainerInterface plaintext, Abstr
    throws SCCException;

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

// Digital Signature
public AbstractSCCSignature sign(AbstractSCCKey key, PlaintextContainerI
    throws SCCException;

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

public AbstractSCCSignature updateSignature(AbstractSCCKey key, Plaintex
    throws SCCException;

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

public boolean validateSignature(AbstractSCCKey key, AbstractSCCSignatur
    throws SCCException;

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

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

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

public boolean validatePasswordHash(PlaintextContainerInterface password
    throws SCCException;

public boolean validatePasswordHash(byte[] password, AbstractSCCPassword
    throws SCCException;
}

abstract interface PlaintextContainerInterface {

public abstract byte[] toBytes();

public abstract String toString(Charset c);

@Override
public abstract String toString();

public abstract boolean validateHash(AbstractSCCHash hash)
    throws SCCException;
```

```
public abstract boolean validatePasswordHash(AbstractSCCPasswordHash pas
    throws SCCException;

public abstract AbstractSCCCiphertext encryptSymmetric(AbstractSCCKey ke
    throws SCCException;

public abstract AbstractSCCCiphertext encryptAsymmetric(AbstractSCCKey k
    throws SCCException;

public abstract AbstractSCCSignature sign(AbstractSCCKey key)
    throws SCCException;

public abstract boolean validateSignature (AbstractSCCSignature signatur
    throws SCCException;

public abstract AbstractSCCHash hash()
    throws SCCException;

public abstract AbstractSCCPasswordHash passwordHash()
    throws SCCException;
}

abstract class AbstractSCCCiphertext {

byte[] msg;

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

public abstract byte[] toBytes();

@Override
public abstract String toString();

public abstract PlaintextContainerInterface decryptSymmetric(AbstractSCC
    throws SCCException;

public abstract PlaintextContainerInterface decryptAsymmetric(AbstractSC
    throws SCCException;

public abstract AbstractSCCCiphertext reEncryptSymmetric(AbstractSCCKey
    throws SCCException;


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

```
abstract class AbstractSCCKey {

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

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

public abstract byte[] toBytes()
    throws SCCException;

public abstract byte[] getPublicKeyBytes()
    throws SCCException;

public abstract byte[] getPrivateKeyBytes()
    throws SCCException;

public abstract KeyType getKeyType();

public abstract String getAlgorithm();
}

abstract class AbstractSCCHash {

byte[] hashMsg;

protected AbstractSCCHash(byte[] hashMsg) {

this.hashMsg = hashMsg;
}

public abstract byte[] toBytes();

@Override
public abstract String toString();

public abstract boolean validateHash(PlaintextContainerInterface plainte
    throws SCCException;

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

public abstract AbstractSCCHash updateHash(PlaintextContainerInterface p
    throws SCCException;
```

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


abstract class AbstractSCCPasswordHash {

byte[] hashMsg;

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

public abstract byte[] toBytes();

@Override
public abstract String toString();

public abstract boolean validatePasswordHash(PlaintextContainerInterface
    throws SCCException;

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

abstract class AbstractSCCSignature {

byte[] signatureMsg;

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

public abstract byte[] toBytes();

@Override
public abstract String toString();

public abstract boolean validateSignature(AbstractSCCKey key)
        throws SCCException;

public abstract AbstractSCCSignature updateSignature(PlaintextContainerI
    throws SCCException;
}
```

Figure 5: Example for a JAVA SCC API

Acknowledgments

*[ ] TODO acknowledge.

Authors' Addresses

Kai Mindermann
iC Consult GmbH

Email: kai.mindermann@ic-consult.com

Lisa Teis

Email: lisateis102@gmail.com