UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Software Engineering

Bachelor's Thesis

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science

Automated Crypto API Rating

by

FURKAN PAMUK

submitted to:
Prof. Dr. Eric Bodden
and
Prof. Dr.-Ing Ben Hermann

Paderborn, Monday 19th October, 2020

DECLARATION

Translation from german:

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Original declaration text in german:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet

Place: Bielefeld Date: 19.10.2020

Signature: [illegible]

Abstract. Today's information system often rely on secure software implemented through Cryptographic Application Programming Interfaces (Crypto APIs). The selection of the most secure cryptographic library can be challenging due to a huge variety of available choices. The definition language *CrySL* allows to define usage rules and patterns for such APIs in a white listing approach. Hence in this bachelor thesis, we present an automated rating system that evaluates the secure usability of libraries through analyzing *CrySL* rules.
Considering properties for *API* usability, we first implemented a set of metrics both to measure the quantitative and qualitative aspects of *CrySL* rules. The analysis was performed on the library rule sets for *BouncyCastle*, *Google Tink* and *JCA*. Our automated evaluation showed that the *Google Tink* library is the most usable choice for the development of secure software systems in the *Java* programming language, followed by *BouncyCastle* and *JCA*.
We concluded that the final rating supports developers to select the secure cryptographic library under a given set of libraries. Our automated rating requires the *CrySL* rule set for respective libraries to be complete, correct and secure. since the rules are still in development, the completeness assumption may not be fulfilled for now. Hence, we emphasize to repeat the evaluation with current rule sets to stay current with the correct *API* choice.

CONTENTS

CHAPTER

1 INTRODUCTION

1.1 Motivation

Today's interconnected society continuously presents new tasks and challenges in digital progress. Moreover, information security systems such as automated border control systems, online banking or even regular smartphones store and transmit more and more sensitive data. Therefore ensuring users' security and privacy is of essential significance. Developing such security systems require dedicated hardware and software components implemented through cryptographic *Application Programming Interfaces (Crypto APIs)*. An *API* is an abstracted set of commands, functions and interfaces enabling users to build software or intercommunicate with external systems. Utilizing and integrating *Crypto APIs* into security software is a must-have prerequisite to protect users' sensitive data and information.
Anderson [And93] points out that the majority of security leaks are not caused by cryptoanalytic or technical weaknesses, but by implementation errors and management failures. Furthermore, Lazar et al. [LCWZ14] figured out, that 83% of the bugs are caused by misuse of cryptographic libraries and the remaining 17% are due to bad construction of the libraries themselves.
Some of the commonly used open source libraries providing cryptographic interfaces, especially for the *Java programming language*, include *Bouncy Castle* [bou], *JCA* [jca] and *Google Tink* [goo]. Determining which of these fit best for secure deployment, can be very challenging. A selection criterion for the choice can be *completeness.* If a developer for instance, wants to encrypt a file, they will prefer an *API* taking care of the whole process, rather than choosing separate ones for file handling and file encryption. Similarly we can look at the *API complexity* and how easy it is to learn its functionalities.

These samples serve as an intuition on how we plan to analyze and evaluate the security of *Crypto APIs*.
The intention of this thesis is to automatically rate the secure usability of *Java Crypto APIs* based on *CrySL* rule specifications [KSA+19], while introducing metrics to measure their security and usability aspects. We will develop an automated and comparable rating between for the rule sets [gita], such that developers get supported at choosing the best fitting *API* for secure software development.

1.2 Outline

To give an overview about the structure of this work, we briefly outline its chapters. Chapter 2 discusses the related work and introduces the research questions. Cryptographic fundamentals, the *CrySL* language [KSA+19] and existing metrics for automated evaluation of language components are introduced in chapter 3. In chapter 4, we propose our metrics to rate and measure various aspects of *CrySL* rules [gita]. Chapter 5 presents the evaluation results of our metrics applied on the *CrySL* rule sets for *Google Tink* [goo], *BouncyCastle* [bou] and *JCA* [jca]. In the sixth and last chapter, we draw a conclusion on the results of this work and discuss future work.

CHAPTER

2 RELATED WORK

There are several tools measuring the usability of *API* automatically. Souza et al. [dSB09] present a tool called Matrix that evaluates *API* usability according to its complexity. It takes the *API* definition, containing all of its classes and methods, as input and computes their complexity resulting in a comparable value. Another automated evaluation tool allowing the comparison of different *APIs* and taking the usage context into consideration is the extensible *API concepts framework* [SK15]. Both these tools still have their limitations. One, for instance, is that measuring the documentation of *APIs* has not been solved yet.

As good and useful the previously mentioned tools are, they are not designed to rate the security of *Crypto APIs*. An important aspect to consider to rate the security of *Crypto APIs* is to inspect the available algorithms in the library for their security level. Since libraries tend to have a diverse set of cryptographic algorithms for different use cases, choosing a sufficiently secure and efficient one, can be a difficult task. For this reason, Jorstad et al. [JL97] introduced metrics to evaluate the strength of cryptographic algorithms. Taking into account, for instance, *the efficiency of existing attacks* on algorithms and their increase in performance with raising computation power, they suggested, that a logical and numerical comparison between algorithms is reasonable. Following this thought of comparability, they defined a set of metrics to measure cryptographic algorithm strength. A sample of these are the *key length, number of rounds, attack efficiency* and *attack complexity* [JL97]. Following this suggestion, we will develop a metric that rates the security level of a *Crypto API* based on the cryptographic strengh of algorithms.

Fundamental for most of the *API* evaluation tools is the work by Weyuker [Wey88]. In this work, he introduced nine metrics to compare and evaluate software complexity mea-

sures in a formal way. Those metrics need to be satisfied by every complexity measure in order to be good and comprehensive [Wey88]. Based on Weyuker's metrics, Misra et al. [MA08] proposed a suite of cognitive metrics to evaluate the complexity of object-oriented systems. These analyze object oriented code features such as *method, attribute, class, inheritance and coupling.* Calculating this suite of metrics for object oriented code bases represents the structural and cognitive complexity of an object oriented system.
In the context of explicitly comparing various *cryptographic libraries*, Acar et al. made an empirical study on functional correctness and security. They also collected users' self-reported sentiment towards the usability of used cryptographic libraries [ABF+17]. Security and usability are interconnected in several ways as they found out. A crucial point is that a *Crypto API* must support features and functionalities for a broad range of cryptographic use cases [ABF+17]. According to them, the inexistence of such features force developers to search for alternative solutions, which can lead to insecure implementations.
Directly related to this topic is the work by Green et al. [GS16], in which they introduce ten principles for creating usable and secure *Crypto APIs.* They state that if cryptographic library developers apply these principles, such as *Making defaults safe and unambiguous* or *Making APIs hard to misuse* [GS16], this could significantly improve security with minimal effort.
Important for our work is the *CrySL* definition language by Krüger et al. [KSA+19]. A *CrySL* rule specification is dedicated to a concrete *Crypto API* and contains usage patterns, such as using correct implementation orders or choosing secure methods, in order to guarantee secure implementations. Through its white listing property [KSA+19], it allows the assumption that the specified rules are correct and secure.

2.1 Contribution

In this work, we will introduce an automatized secure usability rating tool for *Crypto APIs* that analyzes *CrySL* rule specifications [KSA+19]. Since *CrySL* is targeted for the Java programming language, we choose our operating language to be Java as well. As mentioned in the beginning of this chapter, the *CrySL* language allows us to assume that the rules are correct and secure. We further assume the rules are complete, in order to be able to rate the libraries' rule sets.
The goal is to support software developers to integrate the best library for secure software development. For our analysis, we inspect the cryptographic libraries of *Bouncy Castle* [bou], *Google Tink* [goo] and *JCA* [jca]. At the end, we will have a comparable result showing which of the given libraries are best suitable for secure development.

Research Questions

As a guideline, we will introduce our research questions to solve and answer throughout our work.

RQ 1	What properties of *CrySL* specifications are relevant to assess of secure *Crypto API* usability?
RQ 2	What metrics contribute to measure secure usability of *Crypto APIs* based on *CrySL* specifications?
RQ 3	What makes our metrics reasonable for the objective?
RQ 4	Which API fits best for secure development?

For RQ1, we examine *CrySL* rule specifications [KSA+19], concentrating on essential properties for measuring secure *Crypto API* usability. Such properties can be e.g. *memorability or transparency. Memorability*, for instance, indicates how good the developer can remember the correct usage patterns. In a sense, it shows the simplicity of the pattern structure. Similar to this, we plan to investigate on various properties contributing to rate the usability of *Crypto APIs.*

RQ2 presents metrics to establish a comparable rating for *Crypto API* usability. Here, we work on sets of *CrySL* [KSA+19] rule specifications. One important metric, for instance, is the strength and performance of cryptographic algorithms [see Section 4.1]. Moreover, we will focus on the significance of each *CrySL* section [KSA+19], where we prioritize the sections on *secure usability.* To serve as an example, let us consider the following two sections: `ORDER` and `CONSTRAINTS` [KSA+19]. Misusing rules of `CONSTRAINTS` can much simpler cause security threats than misusing the `ORDER` section. To demonstrate this, let us consider the choice of a crytographic algorithm. Assuming that their is an openly available attack e.g. *ROBOT* [BSY18], the prevention to use this algorithm is specified in the `CONSTRAINTS` section.
A wrong implementation order, specified in the `ORDER` section, can cause memory leaks, but requires more sophisticated attacks to leak information. Therefore, the `CONSTRAINTS` section has a higher priority in our rating than `ORDER`. Further details will be explained in Section 4.2.

For RQ3, we compare our metrics to others' following similar goals of supporting secure software development. Thereby, we will validate the right of existence for our metrics [see Section 4.1]. Working on *CrySL* rule specifications, some of the metrics apply differently than others'. The *complexity* property of *APIs*, for instance, generally refers to a variety of functionalities of *APIs*, such as the availability different algorithms for file encryption. Also, as Jorstad et al. [JL97] explained, it can refer to the complexity of a specific cryptographic algorithm. Components of the latter complexity are, for instance, the encryption or decryption type, number of rounds etc. In our case, as mentioned in the answer to **RQ1**, with *complexity* we refer to the pattern variety and quantity of elements in a *CrySL* rule specification [KSA+19]. The goal is to contrast existing metrics to ours', such that one can recognize similarities and differences between various approaches concerning secure *API* usability.

The intention of RQ4 is to interpret the results of the previous research questions. Having built a comparable rating, one should be able to estimate, which of the given set of cryptographic libraries fits best for secure software development. Achieving this result with the introduction of the metrics for its measurement, is the main objective of this work.

CHAPTER

3 BACKGROUND

In this chapter we will explain basic concepts of cryptography, describe relevant *API* properties for our measurment and explain cognitive complexity metrics to evaluate *API* usability. Furthermore we will reintroduce *CrySL*, focusing on the language's sections that specify various usage patterns respectively.

3.1 Terms and Notation

Throughout our analysis we interchangeably use the terms: *CrySL* rule, *CrySL* rule specification, *CrySL* specification and just simply *rule*. All these terms refer to the same thing, which is a `.crysl` file [KSA+19] containing rules and specifications for a specific Java class. Furthermore, we often use the term *secure usability*, which is a usage property that characterizes how user friendly an *API* is structured. More details are explained in Section 3.3.3.

3.2 Cryptography

It can be said that cryptography is an indispensable and crucial field of study, especially for computer science. Historically speaking, it allows secure communication in the presence of malicious third parties, also known as adversaries. More generally, the goal of applying cryptographic methods and algorithms are to prevent adversaries to read secret messages and cause unnecessary damage.
After the introduction of computers most of the known algorithms for secure communication, such as the Vigenère and Caeser cipher, were broken [KL14]. Then the new term

'Modern Cryptography' was introduced [KL14]. Not only does it concern the current algorithms that are used for i.e. message authentication codes, online banking, electronic authentication etc., but it also addresses the potential problems that may rise up in future.
There are two types of encryption schemes, which respectively can be implemented in various ways. These are the *symmetric encryption* and *asymmetric encryption.*
In ***symmetric encryption*** the sender and receiver use the same key both to encrypt and decrypt messages. An important property for *symmetric encryption algorithms* is that they are much more efficient than *asymmetric encryption algorithms.*
In the approach of ***asymmetric encryption*** the key-generation algorithm generates a key-pair, mostly known as *public and private key.* One can use either the generated keys to encrypt the plaintext and then use the other key to decrypt the ciphertext. This provides a more secure way exchange information than *symmetric encryption*, but comes with the cost of lower performance.
Another important type of algorithm, which has the property to compress data of variable length to a fixed (mostly) shorter length and is not reversible, is called a ***hashing algorithm.*** It is known to be deterministic and efficient to compute. The probability of two distinct messages leading to the same hash value is negligibly small and this property is also known as *collision resistance* [BS15]. Use cases for ***hashing algorithms*** are i.e. to check the integrity of data or to securely store and verify the hash values of user passwords.

3.2.1 Common Cryptographic Algorithms

Here, we present common cryptographic algorithms that are or have been widely used in software systems. We will consider all of them in our *Cryptographic Algorithm Security Level (CASL)* metric [see Section 4.1].

Symmetric Algorithms

- **Data Encryption Standard (DES)** is a symmetric-key method for data encryption patented by researchers at IBM. Because of existing attacks, such as brute force or dictionary attacks, this algorithm is known to be obsolete.

- **3DES** is the successor of DES. As an enhanced version of it, it successively applies the DES algorithm three times with three different keys. This results in being inefficient to break with brute force techniques and thus 3DES is in use of various Internet Protocols [AZZ+10].

- **Advanced Encryption Standard (AES)** is another symmetric encryption algorithm, that is known to be more effective than DES and 3DES. It uses a different algorithm and longer key size. AES is used in almost all applications that require encryption.

- **Blowfish** works similar to DES. Unlike DES, it generally uses a larger key size, which needs to be computed first [VP17]. Due to its 64-bit block size, it is vulnerable to birthday attacks, such as SWEET32 [BL16].
- **Twofish** is the successor of Blowfish. What separates it from Blowfish is that it divides the input into 128-bit blocks and uses a key up to 256-bits [SKW$^+$00].

Asymmetric Algorithms

- **RSA** uses a public and a private for its encryption and decryption. Its high level of security is provided by the difficulty of factoring two large prime numbers. It works with large key sizes, such as 1024 or 2048-bits. Over the years, the plain RSA algorithm became vulnerable to, for instance, the *Coppersmith Attack* [Cop01]. Therefore the padded RSA scheme got introduced. While some existing padding schemes, such as *PKCS1.5*, are insecure as well [BSY18], there exist padding schemes that are secure. *RSA-PSS* is an example of a secure padding scheme [Böc11].

Hashing Algorithms

- **MD5** is a very fast hashing algorithm. Due to its small hash output size of 128 bits, it is known to be vulnerable to brute force and birthday attacks.
- **SHA-1** is the Secure Hash Algorithm that outputs a 160 bit Hash output.
- **SHA-2** family of different SHA hashes with different output bit lengths between 224 and 512 bits.
- **SHA-3** is the successor hash family of SHA-1 and SHA-2. It is not susceptible to length extension attacks and provides more security than its predecessors. It differs in its construction, but the output bit length of the Hash values are also between 224 and 512 bits.

3.3 Security Definition and Requirements

In the following, we separate the terms *security* and the meaning of an algorithm to be *secure* from each other, as there is a significant difference.

3.3.1 Information Security

Security in the sense of information security refers to methods, countermeasures and applications that prevent any malicious attacker to access sensitive data. The goal is not only to protect the data itself, but also its environment such as devices, networks and users to have more control. To give an overview of *security goals*, Denis et al. [DJ06] introduced four properties that a security system must have. These are:

1. **Privacy/Confidentiality**

   Ensuring that only the intended receiver can read the sent data.
   This is typically achieved through *symmetric encryption.*

2. **Integrity**

   Ensuring that the sent data is not changed or manipulated during the delivery process from sender to receiver. This is achieved by computing a hash-value of the sent data and combining it with the data.

3. **Authentication**

   Establishing the identity of an individual or the integrity of a message. This can be achieved by entering a password or personal identification number (PIN).

4. **Nonrepudiation**

   Process that ensures that the transferred data is delivered. In other words, it guarantees that neither the sender nor the receiver can deny the delivery of the data. This generally is accomplished through the support of digital signatures, which needs to be signed by an authorized party.

If one can ensure all of the requirements above, a system can be called *secure.*

3.3.2 Computational Security

Kerkhoffs' Principle [Ker] states that an encryption scheme should be made public and what should be kept secret are the secret keys being used. Many security systems are breakable in theory or at least are not considered to be completely secure. Successfully breaking any of them cannot be done in "reasonable time with any reasonable probability success" as Katz et al. [KL14] said. It is sufficient to say that an encryption scheme is not breakable, if even with the fastest accessible supercomputer, the probability of breaking the encryption scheme is not better than $\mathbf{10^{-30}}$ in **200** years computing time [KL14]. In other words, if we *cannot* break an encryption scheme with probability better than $\mathbf{10^{-30}}$ in **200** years computing time, it is called *secure.* We will use this term of security to determine the security level of cryptographic algorithms [see Section 4.1].

3.3.3 Secure Usability of APIs

To better understand our interpretation of the term *secure usability* in the context of *APIs*, we refer to the union of *information security* [see Section 3.3.1] and *computational security* [see Section 3.3.2]. The goal is both to fulfill as many *security goals* [see Section 3.3.1] as possible and use cryptographic algorithms that provide computational security. In order to guarantee *secure usability*, there exist textual specification languages, such as *CrySL*, which we will explain in Section 3.4.2.
Also contributing to *secure usability* is the usability of *APIs*, since this allows to measure

the users' convenience with the *API*. In security related tasks, such as developing secure software, this is of essential significance. The more convenient a developer is with the used *API*, the more he or she is aware of the result and tends to cause lesser security threats. Hilbert et al. [HR00] stated in their work that *usability* consists of multiple components and is associated with *Learnability, Efficiency, Memorability, Errors and Satisfaction.*

3.4 Cryptographic Application Programming Interface

In order to explain the term *Cryptographic API (Crypto API)*, we need to recall the term: *Application Program Interface (API)*. An *API* is an abstracted set of commands, functions and interfaces enabling users to build software or intercommunicate with external systems. Making use of common *APIs* in various fields simplifies software development, as it allows code reusing and accelerates the learning process. Today, there are *APIs* for almost every imaginable category such as travel, music and cryptography. *Crypto APIs* are designed to specify interfaces for performing cryptographic operations such as encryption, decryption, hashing, signature generation and verification. Throughout our work, we refer to the terms *Crypto API* and cryptographic library.

3.4.1 Java Cryptography Architecture (JCA)

Throughout our investigation on measuring the secure usability of *Crypto APIs*, we will use the *Java* programming language including the *Java Cryptography Architecture (JCA)* and the associated *Java Cryptography Extension (JCE)* [jca] packages. *JCA* is the *Crypto API* of *Java* and its design concept allows developers to integrate various security operations and functionalities into their software. The two core concepts of *JCA* are built around *Implementation independence* & *interoperability* and *algorithm independence* & *extensibility* [jca]. It contains security functionalities, such as various encryption methods, key generation, hashing etc. Following a provider based structure, in which the user, for instance, is able to use the `Cipher` class for file encryption or decryption, the concrete implementation of that class (i.e. used algorithms) depends on the selected *provider* [jca].

3.4.2 CrySL: Validating Correct Usage of Crypto APIs

There is a huge variety of research about designing textual specification languages that ensure correct API usage [KSA+19]. *CrySL* is a definition language allowing cryptography experts to define secure usage rules and patterns for respective *APIs* in a lightweight special-purpose syntax [KSA+19]. Other than most other definition languages that follow a *black listing* approach, by specifying forbidden usage patterns, the design is to *white list* correct usage patterns to obtain concise usage specifications [KSA+19]. The language also stands out for its structure of following simple and familiar programming concepts that most cryptography developers can learn without large obstacles. This is facilitated by its language rules and patterns specified as individual classes or as it is called *sections*, each of them following a unique purpose. Let us look at the following example of a *CrySL* rule in Figure 3.1

```
SPEC javax.crypto.KeyGenerator

OBJECTS
  java.lang.String algorithm;
  int keySize;
  javax.crypto.SecretKey key;

EVENTS
  g1: getInstance(algorithm);
  g2: getInstance(algorithm, _);
  GetInstance := g1 | g2;

  i1: init(keySize);
  i2: init(keySize, _);
  i3: init(_);
  i4: init(_, _);
  Init := i1 | i2 | i3 | i4;

  GenKey: key = generateKey();

ORDER
  GetInstance, Init?, GenKey

CONSTRAINTS
  algorithm in {"AES", "Blowfish"};
  algorithm in {"AES"} => keySize in {128, 192
      256};
  algorithm in {"Blowfish"} => keySize in {128
      192, 256, 320, 384, 448};

ENSURES
  generatedKey[key, algorithm];
```

Figure 3.1: CrySL rule for javax.crypto.KeyGenerator [KSA+19]

The *CrySL* rule presented in Figure 3.1 is specified for the `KeyGenerator` class, as the `SPEC` section [KSA+19] indicates. Subsequently, the `OBJECTS` [KSA+19] section contains the *Java* variables with respective data types for allowed usage. Note that, if the user tries to use any other variable not contained in `OBJECTS` section, the static analyzer *CogniCrypt* [KNR+17] throws an error.
`EVENTS` [KSA+19] defines usage patterns and methods for secure implementations. For instance, if there would exist a method `init(keySize, algorithm, key)` in the *API*, its usage would be denied due to missing specification in the rule [see Figure 3.1]. The `ORDER` section [KSA+19] basically combines the previous sections by specifying how the elements in the `EVENTS` section are allowed to be used. The pattern: `GetInstance, Init?, GenKey` [see Figure 3.1] indicates that we must start with `GetInstance`, follow

with an optional `Init` and terminate with a `GenKey`.
The `CONSTRAINTS` section [KSA+19] specifies that the variable `algorithm` must be *AES* or *Blowfish*. Depending on the chosen `algorithm` its `keySize` must be chosen respectively. The `ENSURES` section [KSA+19] guarantees the secure predicate `generatedKey[key, algorithm]`, if all sections are used correctly. Predicates can be required by another rule within the library's rule set. *CrySL* rules have mandatory and optional sections [KSA+19]. In the following we present these sections and give an example for each.

Mandatory Sections (Classes)

`OBJECTS`

This section defines all the allowed objects that are usually used as parameters or return values in the `EVENTS` section. In Figure 3.1, `OBJECTS` allows the elements: `algorithm, keySize` and `key` with their corresponding data types. This prescribes that the specified class `KeyGenerator` is allowed to be used only with these variables.

`EVENTS`

The purpose of this section is to define all methods with respective parameters that contribute to a correct usage of the specified class. Note, that a method used with wrong parameters would cause a potential threat to its security. Let us consider `Init` [see Figure 3.1] as an example. `Init` can only take the values of `i1,i2,i3` or `i4` for correct implementation.

`ORDER`

In this section, the patterns specified in the previous `EVENTS` section are used within a regular expression. The goal is to define a secure usage pattern including *order of execution, amount of possible method calls* etc., obtaining a valid usage pattern.
We demonstrate an example with the pattern: `((a|b),c)`. Assuming that `a,b` and `c` are valid predicates defined in the `EVENTS` section, we have the following order. The user must start with either `a` or *b*. Subsequent to that is `c`, which terminates the pattern.

`CONSTRAINTS`

This section uses the elements of `OBJECTS` as parameter or return values in the `EVENTS` section. Moreover, it sets constraints to, for instance, choosing only a subset of available algorithms and specific keysizes for respective algorithms in conjunction with i.e. a mode of encryption.
To demonstrate this, let us consider the `CONSTRAINTS` section of Figure 3.1. Here the user is only allowed to set the variable `algorithm` to be `AES` or `Blowish`. If `AES` is chosen, the `keySize` variable must be **128, 192 or 256 bit** and if `algorithm` is `Blowfish`, then `keySize` must be **128, 192, 320, 384 or 448 bit**.

`ENSURES`

This section represents the guarantee of the specified class, if it used properly according to the sections. For example, the *KeyGenerator* rule in Figure 3.1 defines a predicate `generatedKey` with the *generated key object* and *parameters* (Lines 37-38).

Optional Sections (Classes)

`REQUIRES`

This section requires the specified predicate [KSA+19] guaranteed in the `ENSURES` section of another *CrySL* rule within the rule set. Assume that we have a rule distinct from `KeyGenerator`, requiring the *generatedKey* predicate from the `ENSURES` section of the `KeyGenerator` rule. This would be a valid element for the `REQUIRES` section.

`FORBIDDEN`

This section specifies methods that must not be called. One can provide an alternative to that forbidden method such that the developer is presented a secure solution. To illustrate this, in Figure 3.2 Line 72 the `PBEKeySpec(char[])` element is considered to be insecure. As a valid alternative, we can use the `create` element, which is defined in the `EVENTS` section of the same rule.

`NEGATES`

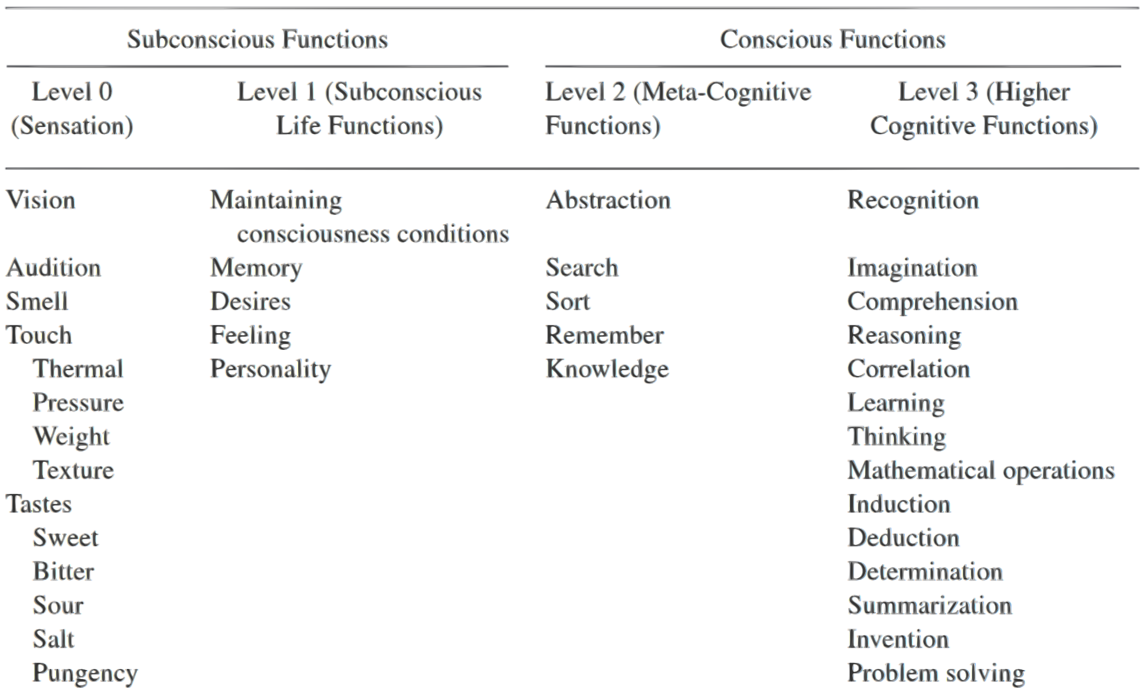
The `NEGATES` section provides the functionality of deleting or rather killing a predicate. We can see this in Line 83 of Figure 3.2. There, the predicate `keyspec` with *its generated object* and *parameters* is killed [KSA+19] and cannot be used anymore.

```
SPEC javax.crypto.spec.PBEKeySpec

OBJECTS
  char[] pw;
  byte[] salt;
  int it;
  int keylength;

EVENTS
  create: PBEKeySpec(pw, salt, it, keylength);
  clear: clearPassword();

FORBIDDEN
  PBEKeySpec(char[]) => create;
  PBEKeySpec(char[],byte[],int) => create;

ORDER
  create,  clear
...

ENSURES
  keyspec[this, keylength] after create;

NEGATES
  keyspec[this, _];
```

Figure 3.2: CrySL rule for javax.crypto.spec.PBEKeySpec [KSA+19]

3.5 Cognitive Complexity

Cognitive Complexity is a measuring concept of the cognitive and psychological software complexity for human user interactions [SW03]. Through the concept of the *cognitive weight*, defining the degree of difficulty or relative time and effort for comprehending a given software component, Shao et al. [SW03] introduced new so called *Cognitive Complexity Metrics*. These metrics can be used to obtain critical information about reliability and maintainability of software systems [MAFSD18]. To get an overview on the levels of cognitive phenomena, we refer to Figure 3.3, presented by Wang [Wan03] in the following.



Subconscious Functions		Conscious Functions	
Level 0 (Sensation)	Level 1 (Subconscious Life Functions)	Level 2 (Meta-Cognitive Functions)	Level 3 (Higher Cognitive Functions)
Vision	Maintaining consciousness conditions	Abstraction	Recognition
Audition	Memory	Search	Imagination
Smell	Desires	Sort	Comprehension
Touch	Feeling	Remember	Reasoning
Thermal	Personality	Knowledge	Correlation
Pressure			Learning
Weight			Thinking
Texture			Mathematical operations
Tastes			Induction
Sweet			Deduction
Bitter			Determination
Sour			Summarization
Salt			Invention
Pungency			Problem solving

Figure 3.3: Classification of cognitive phenomena [Wan03]

The cognitive efforts and difficulties are divided into four levels. *level 0 and 1* are considered as *subconscious functions*, meaning that the cognitive effort it takes to, for instance smell or feel confident to do something, is relatively small. *Level 3 and 4* are so called *conscious functions* [see Figure 3.3]. Unlike *subconscious functions*, these are cognitively more challenging depending on the performed task, such as remembering or reasoning (s. Figure 3.3). Note, that the higher the *level* gets, the more cognitive effort it takes to accomplish the task. Therefore, *APIs* with lower levels of cognitive difficulty, such as fewer elements to remember, make developers feel more confident implementing software components and prevents erroneous software products.

3.6 Existing Cognitive Complexity Metrics

In this section, we review existing cognitive complexity metrics. For the most part, we base our metrics [see Section 4.1] on analogous concepts of measuring quantitative aspects of *APIs*, but adapted on *CrySL* rule specifications.

Attribute Complexity (AC)

This metric represents the complexity due to data members (attributes) by taking the total number of attributes associated with a class [MAFSD18].
Its formula is:

$$\boldsymbol{AC = \sum_{p=1}^{n} 1} \qquad \text{, where } \boldsymbol{n} \text{ is the total number of attributes}$$

It is used to measure the quantity of attributes assigned in a class. For demonstration, let us consider the *Java* class `Car`:

```
public class Car {
    private String model;
    private int year, hp;
    private double price;
}
```

Here, we have $\boldsymbol{AC}_{\texttt{Car}} \boldsymbol{= 4}$. Having multiple *Attribute Complexity (AC)* values, we can compare their complexity easily. The smaller the *AC* value is, the lower and better is its complexity.

Average Attribute Complexity per Class (AAC)

This metric represents the average *Attribute Complexity (AC)* [see Section 3.6] per class [MAFSD18]. Its formula is:

$$\boldsymbol{AAC = \frac{1}{m}\sum_{i=1}^{m} AC} \qquad \text{, where } \boldsymbol{m} \text{ is the total number of classes}$$

Measuring the attribute complexity provides a good foundation to implement our metrics to rate the *secure usability CrySL* rules. The goal is to measure the quantity of data members, which provide good comparability.

3.7 Existing Cryptographic Algorithm Metrics

Jorstad et al. [JL97] introduced characteristics for evaluating the strength of cryptographic algorithms. Through considering, for instance, *the efficiency of existing attacks* on algorithms and their increase in attack performance with raising computation power, they suggested that a logical and numerical comparison between algorithms seems to be intuitive [JL97]. Following this thought of comparability, they proposed metrics to measure cryptographic algorithm strength.
In order to rank specific algorithms according to their strength, we list the relevant metrics for our analysis [JL97].

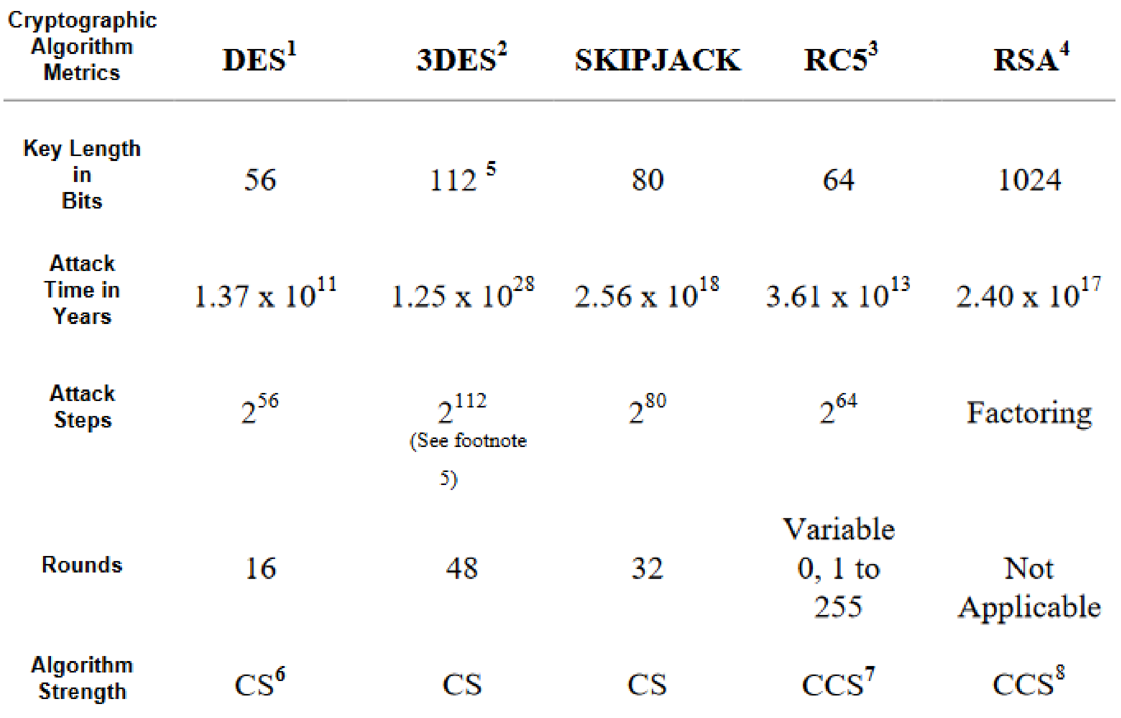
1. **Key Length Metric** states that the bigger the used *key length* is, the more security it offers, since the attempts to find the correct key increases with increasing *key length.*

2. **Attack Steps Metric** is defined by the number of steps to perform the best known attack, which helps to determine the time to perform the attack on a particular processor.

3. **Attack Time Metric** is defined by the time it takes to perform the best known attack on a specific processor.

4. **Rounds Metric** are targeted only for some algorithms, that use rounds in their implementations i.e. *DES.* The point is that the more rounds an algorithm has, the more security does it provide in most cases.

5. **Algorithm Strength** requires an objective, numeric key length in order to compare respective algorithms witch each other. With the *Suggested Algorithm Strength Evaluation Critera* and the *Suggested Algorithm Strength Scale Graduations* [JL97], Jorstad et al. presented an example application of their proposed measuring techniques and obtained the result, shown in Figure 3.4. Note, that this metric is basically an evaluation according to the suggested graduation mentioned previously.



Cryptographic Algorithm Metrics	DES[1]	3DES[2]	SKIPJACK	RC5[3]	RSA[4]
Key Length in Bits	56	112 [5]	80	64	1024
Attack Time in Years	1.37×10^{11}	1.25×10^{28}	2.56×10^{18}	3.61×10^{13}	2.40×10^{17}
Attack Steps	2^{56}	2^{112} (See footnote 5)	2^{80}	2^{64}	Factoring
Rounds	16	48	32	Variable 0, 1 to 255	Not Applicable
Algorithm Strength	CS[6]	CS	CS	CCS[7]	CCS[8]

Figure 3.4: Pilot Examples of Metrics for Cryptographic Algorithms [JL97]

Following the concepts of grading algorithms based on their cryptogaphic strength by Jorstad et al. [JL97], we will introduce our metric *Cryptogaphic Algoritm Security Level* [see Section 4.1] to determine the security level of a *CrySL* rule.

CHAPTER

4 ANALYSIS CONCEPTS

The goal of this chapter is to investigate on the second and third research question. We will develop metrics to measure *CrySL* rule specifications based on the API properties *Memorability, Learnability, Errors* and *Satisfaction* [HR00]. Simultaneously, we will explain how the metrics will be assessed for our evaluation.

4.1 Proposed Metrics To Analyze CrySL Specifications

In the following we will present our metrics to evaluate *CrySL* rule specifications. Inspired by existing metrics to analyze *Cognitive Complexity* of software systems [see Section 3.5, Section 3.6] and *Strength of Cryptographic Algorithms* [see Section 3.7], we have designed our own metrics to evaluate the secure usability of *CrySL specifications* [KSA$^+$19].
Following the five subparts of system usability [HR00] and considering the security principles by Green et al [GS16], we present our metrics in the following:

Definitions and Notation

let $\boldsymbol{R}$ be the set of all CrySL rule specifications for a given library,
let $\boldsymbol{A}$ be the set of cryptographic algorithms present in one $\boldsymbol{r} \in \boldsymbol{R}$,
let $\boldsymbol{S = \{1, 2, 3, 4, 5\}}$ be the set of graduation assignments

Elements Per Section (EPS)

This metric represents the number of elements in an arbitrary *CrySL class* [see Section 3.4.2]. An element is generally specified in a separate line, except in the `ORDER` section. There we consider the number of states [KSA+19].
More formally we have:

$$\boldsymbol{EPS = \sum_{i=1}^{n} 1} \qquad \text{, where } \boldsymbol{n} \text{ is the total number of elements in a CrySL section}$$

i.e. in Figure 3.1 for the sections `CONSTRAINTS` and `ORDER`, we have:

$$\boldsymbol{EPS_{CONSTRAINTS} = 3}$$

Note, that for $\boldsymbol{EPS_{ORDER}}$ we count the state nodes, as shown in Figure 4.1 below and not the specified line: `GetInstance, Init?, GenKey`.
Thus, we have:

$$\boldsymbol{EPS_{ORDER} = 4}$$

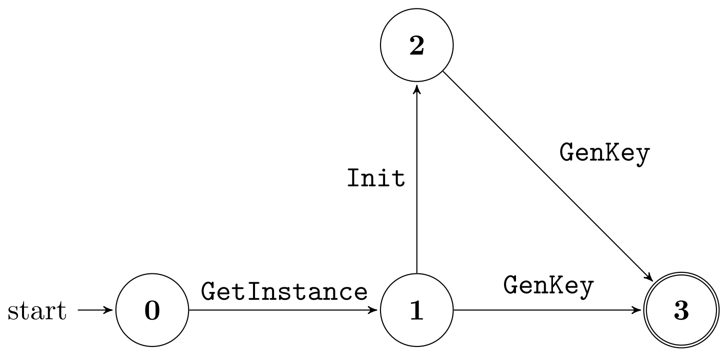


Figure 4.1: State Machine for `ORDER` pattern: `GetInstance, Init?, GenKey`

This is a quantitative metric measuring the *CrySL* rule complexity. It calculates the number of restrictions that are specified within a *CrySL* section (class). Considering the API properties *Memorability* and *Learnability* [see Section 3.3.3], a higher value in *EPS* increases the cognitive level [see Figure 3.3] and decreases its simplicity [Dou08]. Therefore, a lower *EPS* value contributes to better *secure usability.*

Elements Per Rule (EPR)

Similar to the previous *EPS* metric, *EPR* represents the total number of elements in a *CrySL rule* [see Section 3.4.2]. Here, we again count the number of states for the `ORDER` section.

Formally we have:

$$\boldsymbol{EPR} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \mathbf{1} \qquad \text{, where } \boldsymbol{n} \text{ is the total number of elements in a CrySL rule}$$

i.e. in Figure 3.1, we have:

$$\boldsymbol{EPR}_{KeyGenerator} = \mathbf{20}$$

Following the similar goals of measuring quantitative properties just as the *Elements per section (EPS)* metric, we want to treat the *CrySL* rule as a whole and calculate its total number of elements specified. Unlike *EPS*, we have an absolute value here that can be related to the whole set $\boldsymbol{R}$. This is accomplished in the *AEPR* metric later.

Parameters Per Method (PPM)

PPM measures the quantitative complexity of a method by counting the number of required parameters a method uses. This metric is only dedicated to the `EVENTS` section of a rule [see Section 3.4.2]. *CrySL* allows to specify a placeholder parameter, denoted as the underscore symbol: "_" [KSA+19], permitting any parameter type that fits into this pattern. *PPM* treats the placeholder symbol as a regular parameter, since from a cognitive complexity standpoint the effort of memorizing, whether its an underscore or a specific parameter, is equal [see Section 3.5]. We demonstrate this idea with the help of the following two methods:

1. `RSAPrivateCrtKeyParameters(modulus, _, _, _, p, q, _, _, _)`
2. `RSAPrivateCrtKeyParameters(modulus, a, b, c, p, q, d, e, f)`

In both methods, the user has to know the concrete parameter positions that are not flexible, or in other words are not the placeholder symbol "_". Even though the second method has more not flexible parameters, the effort to remember whether all parameters or three parameters are not flexible is cognitively the same [see Section 3.5]. Formally, we have:

$$\boldsymbol{PPM} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \mathbf{1} \qquad \text{, where n is the total number of parameters in method}$$

In the above example, we have the values:

$$\boldsymbol{PPM_1} = \mathbf{8}$$
$$\boldsymbol{PPM_2} = \mathbf{8}$$

Therefore, any element (parameter) in the `EVENTS` section is treated equally and counts as an additional element. As Bloch [Blo06] suggested to avoid long parameter list, we follow the idea that the smaller the *PPM* value is, the better its *secure usability* gets.

Number of Forbidden Methods (NFM)

This metric measures the number of forbidden methods in a *CrySL* rule. The information is extracted from the `FORBIDDEN` section [KSA+19] and the `noCallTo` helper functions [KSA+19]. The goal is to measure how many methods of the library are always insecure and cannot be made secure by any restrictions.
Let $\boldsymbol{U}$ be the union of the methods specified in the `FORBIDDEN` section and the `noCallTo` helper methods. Hence, we formally have:

$$\boldsymbol{NFM} = \sum_{\boldsymbol{i \in U}} \mathbf{1} \qquad \text{, where n is the total number of methods in a rule}$$

To demonstrate this, let us consider a rule $\boldsymbol{r} \in \boldsymbol{R}$ containing the following elements in the `FORBIDDEN` section [KSA+19] and the helper method *noCallTo*:

1. `PBEKeySpec(char[])`
2. `PBEKeySpec(char[],byte[],int)`
3. `noCallTo[init(encmode, cert)]`
4. `noCallTo[init(encmode, key)]`

Thus, we have $\boldsymbol{NFM_r} = \mathbf{4}$, since each element is treated and counted equally. A rule having lesser forbidden methods or in other words having a smaller $\boldsymbol{NFM}$ value, indicates better *secure usability* for developers.

Accepted Orders (AO)

The following metric shows how many possible implementation options a *CrySL* specification has. *AO* is applied dedicated to the `ORDER` section [KSA+19]. In some sense, we can say that this represents the variety of the specified class. We have the formula:

$$\boldsymbol{AO} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \mathbf{1} \qquad \text{, where } \boldsymbol{n} \text{ is the total number of accepted states [KSA+19]}$$

i.e. in Line 76 of Figure 3.2, we have:

$$\boldsymbol{AO}_{PBEKeySpec} = \mathbf{1} \quad \text{[see Figure 4.2]}$$

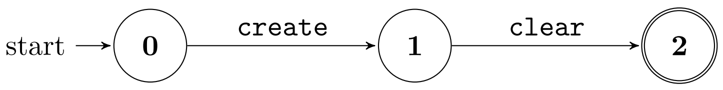


Figure 4.2: State Machine for `ORDER` pattern: `create, clear`

To demonstrate another example, if we had `(create | clear)*` instead of `create, clear` in Line 76 of Figure 3.2, then we would also have:

$$\boldsymbol{AO}_{PBEKeySpec} = \mathbf{1} \quad \text{[see Figure 4.3]}$$

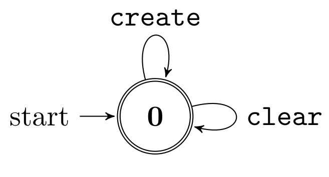


Figure 4.3: State Machine for `ORDER` pattern: `(create | clear)*`

We observe that this metric on its own can lead to ambiguous results. Therefore, we will consider the percentage of accepting states over all available states.

Ratio of Accepted Orders (RAO)

With RAO we represent the relation of accepted states to total amount of states [KSA+19]. In other words, without changing the specified usage order and using only secure methods from the `EVENTS` section, what is the percentage a user could implement insecurely by, for instance, missing to clear the memory after working on sensitive data.
Formally, we have:

$$\boldsymbol{RAO} = \frac{\boldsymbol{AO}}{\boldsymbol{n}} \qquad \text{, where } \boldsymbol{n} \text{ is the total number of states [KSA+19]}$$

We will again consider the previous example to illustrate the idea.
In Line 76 of Figure 3.2, we have the consecutive usage pattern: `create, clear`.
Thus,

$$\boldsymbol{AO}_{PBEKeySpec} = \mathbf{1} \text{ and } \boldsymbol{n} = \mathbf{3} \quad \text{[see Figure 4.2]}$$

and:

$$\boldsymbol{RAO}_{PBEKeySpec} = \frac{\mathbf{1}}{\mathbf{3}} \approx \mathbf{33.33\%}$$

This $\boldsymbol{RAO}_{PBEKeySpec}$ states that our variability in varying the implementation is weak, since we have a strict order sequence here.
Now, let us consider the `ORDER` pattern: `(create | clear)*`. This results in:

$$\boldsymbol{AO}_{PBEKeySpec} = \mathbf{1} \text{ and } \boldsymbol{n} = \mathbf{1} \quad \text{[see Figure 4.3]}$$

and therefore:

$$\boldsymbol{RAO}_{PBEKeySpec} = \frac{\mathbf{1}}{\mathbf{1}} = \mathbf{100\%}$$

This $\boldsymbol{RAO}_{PBEKeySpec}$ states that our variability is very strong, since all possible states that can be reached are accepting, meaning that their implementation is valid and secure.

As stated previously, this metric represents the error probability despite using secure methods in correct order. Obviously the higher the percentage is, the better the secure usability gets, since the cognitive effort is reduced due to more variety.

Average Elements Per Rule (AEPR)

This metric calculates the average elements per rule *(EPR)* over the whole set of rules $\boldsymbol{R}$. *AEPR* measures the general complexity of the library's rule set, showing how many elements the average rule has. Computing the average applies well for our use case, because it considers all rules specified in the whole rule set. Formally we have:

$$\boldsymbol{AEPR = \frac{1}{n} \sum_{r \in R} EPR_r} \qquad \text{, where } \boldsymbol{n} \text{ is the total number rules in a library}$$

Let us assume that the library rule set $\boldsymbol{R}$ consists of the two rules presented in Figure 3.1 and Figure 3.2. Thus, we have:

$$\boldsymbol{EPR_{\mathit{PBEKeySpec}} = 13}$$
$$\boldsymbol{EPR_{\mathit{KeyGenerator}} = 20}$$

, which leads to:

$$\boldsymbol{AEPR_R = \frac{13 + 20}{2} = 16.5}$$

The rule set $\boldsymbol{R}$ for the specified library has an $\boldsymbol{AEPR_R}$ value of **16.5**. The assessment follows the same principle of the *EPR* metric, in which a smaller value indicates better *secure usability.*

Average Elements Per Section (AEPS)

This metric represents the average number of elements per section *(EPS)* of a CrySL rule specification $\boldsymbol{r \in R}$. The value of this metric, analogous to *EPR*, is representative for a single *CrySL* rule and shows the distribution density of the rules. Calculating the average, we only consider specified sections, such that empty sections do not contribute to the result.
For $\boldsymbol{n}$ being the total number of CrySL sections in a rule $\boldsymbol{r \in R}$ and $\boldsymbol{i}$ being the $\boldsymbol{i}$-th section of $\boldsymbol{r}$, we have:

$$\boldsymbol{AEPS = \frac{1}{n} \sum_{i=1}^{n} EPS_i}$$

i.e. in Figure 3.1 for the *KeyGenerator* rule, we have:

$$\begin{aligned}
&\boldsymbol{AEPS}_{KeyGenerator} \\
&= \frac{\mathbf{1}}{\mathbf{5}}(\boldsymbol{EPS_{Objects}} + \boldsymbol{EPS_{Events}} + \boldsymbol{EPS_{Order}} + \boldsymbol{EPS_{Constraints}} + \boldsymbol{EPS_{Ensures}}) \\
&= \frac{\mathbf{1}}{\mathbf{5}}(\mathbf{3} + \mathbf{8} + \mathbf{4} + \mathbf{3} + \mathbf{1}) \\
&= \mathbf{3.8}
\end{aligned}$$

The CrySL rule of the Java Class *KeyGenerator* has **3.8** elements per section in average. Similar to the previous metrics, a smaller value for *AEPS* indicates better *secure usability.*

Average AEPS over all rules (AAEPS)

With this metric we represent the average *AEPS* value over all rules $\boldsymbol{r} \in \boldsymbol{R}$. The approach is analogous to *AEPR*, in which we compute a value for the whole library. For $\boldsymbol{n}$ being the total number of CrySL rules in the respective library and $\boldsymbol{i}$ being the $\boldsymbol{i}$-th rule, we have:

$$\boldsymbol{AAEPS} = \frac{\mathbf{1}}{\boldsymbol{n}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{AEPS_i}$$

To demonstrate this, assume that a library rule set $\boldsymbol{R_x}$ consists of three distinct *CrySL* rules with the following *AEPS* values:

$$\begin{aligned}
\boldsymbol{AEPS_1} &= \mathbf{3.8} \\
\boldsymbol{AEPS_2} &= \mathbf{4.0} \\
\boldsymbol{AEPS_3} &= \mathbf{4.2}
\end{aligned}$$

Then, we have:

$$\begin{aligned}
\boldsymbol{AAEPS_{R_x}} &= \frac{\mathbf{3.8} + \mathbf{4.0} + \mathbf{4.2}}{\mathbf{3}} \\
&= \mathbf{4.0}
\end{aligned}$$

Having $\boldsymbol{AAEPS_{R_x}}$ equal to **4.0**, we know that the average rule $\boldsymbol{r_x} \in \boldsymbol{R_x}$ has **4.0** elements per section. Here again, the smaller the value is, the better is its *secure usabiltiy.*

Average Parameters per Method (APPM)

This metric represents the average parameters per method (PPM) of a CrySL rule specification. Again, a smaller value indicates better *secure usability*. For $\boldsymbol{n}$ being the total number of elements in the `EVENTS` section and $\boldsymbol{i}$ being the $\boldsymbol{i}$-th element, we have:

$$\boldsymbol{APPM = \frac{1}{n}\sum_{i=1}^{n} PPM_i}$$

Let us consider a rule $\boldsymbol{r \in R}$ containing only the following three methods:

1. `method1(a, b)`
2. `method2(a, _, _)`
3. `method3(a, b, c, d)`

$$\begin{aligned}&\boldsymbol{APPM_R}\\&\boldsymbol{= \frac{1}{3}(PPM_{method1} + PPM_{method2} + PPM_{method3})}\\&\boldsymbol{= \frac{1}{3}(2+3+4)}\\&\boldsymbol{= 3}\end{aligned}$$

Here, the rule $\boldsymbol{r \in R}$ has three parameters per method in average.

Average APPM over all rules (AAPPM)

This metric gives an overview over the whole library and represent the average *APPM* over all rules in the library. For $\boldsymbol{n}$ being the total number of CrySL rules in the respective library and $\boldsymbol{i}$ being the $\boldsymbol{i}$-th rule, we have:

$$\boldsymbol{AAPPM = \frac{1}{n}\sum_{i=1}^{n} APPM_i}$$

Here, we left out the example, since the approach of calculating the average over the whole rule set $\boldsymbol{R}$ is analogous to *AEPR* and *AAEPS*. Again, the smaller the *AAPPM* value is, the better is the *secure usability*.

All Number of Forbidden Methods (ANFM)

This metric measures the total number of forbidden methods over all rules in the library. We add up all $\boldsymbol{NFM}$ values to show how many faulty and vulnerable methods exist in the given library. For $\boldsymbol{n}$ being the total number of CrySL rules in the respective library rule set $\boldsymbol{R}$ and $\boldsymbol{i}$ being the $\boldsymbol{i}$-th rule, we have:

$$\boldsymbol{ANFM = \sum_{i=1}^{n} NFM_i}$$

Assume that our rule set $\boldsymbol{R}$ consists of the rules $\boldsymbol{a, b}$ and $\boldsymbol{c}$. Further assume the following:

$$\begin{aligned}\boldsymbol{NFM_a} &= \boldsymbol{1}\\ \boldsymbol{NFM_b} &= \boldsymbol{2}\\ \boldsymbol{NFM_c} &= \boldsymbol{3}\end{aligned}$$

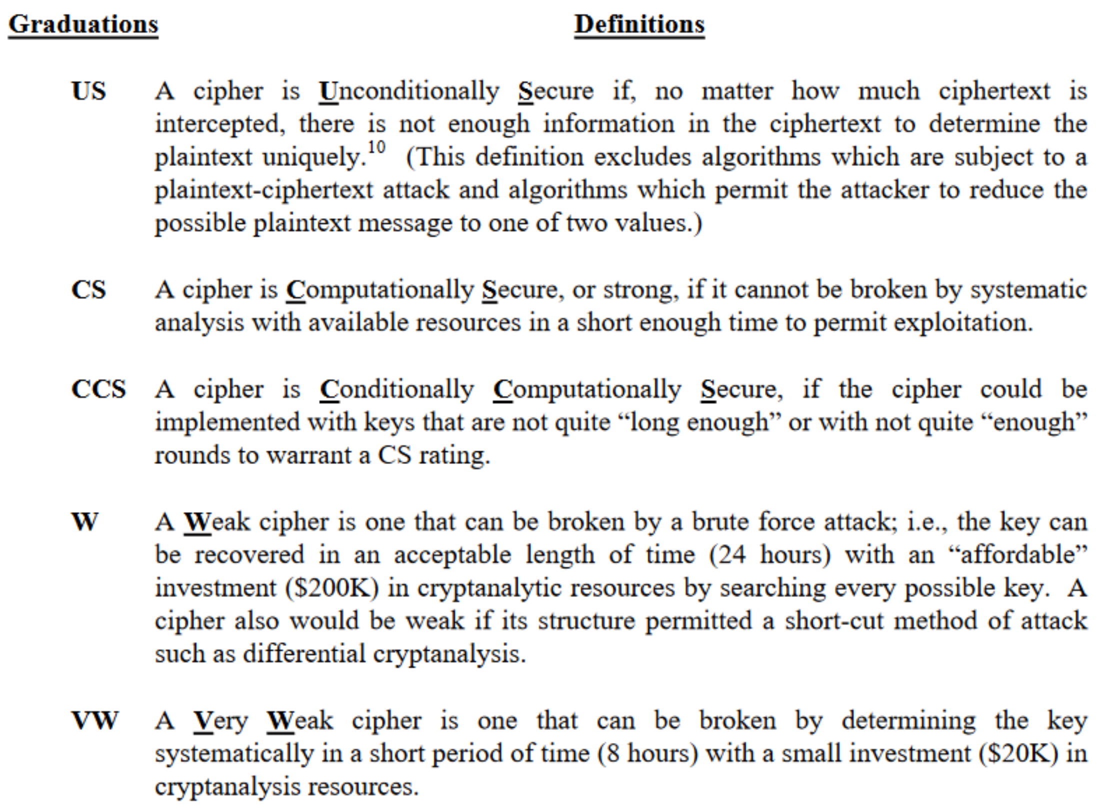
Thus, we have:

$$\begin{aligned}\boldsymbol{ANFM_R} &= \boldsymbol{NFM_a + NFM_b + NFM_c}\\ &= \boldsymbol{1+2+3}\\ &= \boldsymbol{6}\end{aligned}$$

Having six vulnerable methods, one should remove these from the library or redesign them to prevent security threats. As mentioned previously, the smaller the $\boldsymbol{ANFM}$ value is, the better is the *secure usability.*

Cryptographic Algorithm Security

Based on the metrics to measure cryptographic algorithm strengh, as mentioned in Section 3.7, we will rank cryptographic algorithms based on their security level. We will determine the security level of an algorithm by following the *suggested graduations* by Jorstad et al. [JL97]. These are the illustrated in the following Figure 4.4.



Graduations	Definitions
US	A cipher is Unconditionally Secure if, no matter how much ciphertext is intercepted, there is not enough information in the ciphertext to determine the plaintext uniquely.[10] (This definition excludes algorithms which are subject to a plaintext-ciphertext attack and algorithms which permit the attacker to reduce the possible plaintext message to one of two values.)
CS	A cipher is Computationally Secure, or strong, if it cannot be broken by systematic analysis with available resources in a short enough time to permit exploitation.
CCS	A cipher is Conditionally Computationally Secure, if the cipher could be implemented with keys that are not quite "long enough" or with not quite "enough" rounds to warrant a CS rating.
W	A Weak cipher is one that can be broken by a brute force attack; i.e., the key can be recovered in an acceptable length of time (24 hours) with an "affordable" investment ($200K) in cryptanalytic resources by searching every possible key. A cipher also would be weak if its structure permitted a short-cut method of attack such as differential cryptanalysis.
VW	A Very Weak cipher is one that can be broken by determining the key systematically in a short period of time (8 hours) with a small investment ($20K) in cryptanalysis resources.

Figure 4.4: Five Subjective Graduation Algorithm Strength Scale [JL97]

For our measurement, we assign those graduations the equivalent numeric values in Table 4.1.

Graduation	Numeric Value
VW *(Very Weak)*	1
W *(Weak)*	2
CCS *(Conditionally Computationally Secure)*	3
CS *(Computationally Secure)*	4
US *(Unconditionally Secure)*	5

Table 4.1: Graduation Assignments

We define graduation function $\boldsymbol{grad}$ as follows:

$$\boldsymbol{grad(a) = s}, \text{ where } \boldsymbol{s \in S, a \in A.}$$

Cryptographic Algorithm Security Level (CASL)

This metric represents the security level of a *CrySL* rule specification. It extracts the algorithms out of the `CONSTRAINTS` section [KSA+19] and calculates the security level by taking the most insecure algorithm and take this as security level. Since the developer is allowed to use the most insecure algorithm that is specified within a rule, the security level of the rule must be a lower bound, which is determined by the minimum of all security levels. The graduations are based on current research and can continuously be updated. We define the security level function $\boldsymbol{CASL}$ of a CrySL rule $\boldsymbol{r} \in \boldsymbol{R}$ as follows:

$$\boldsymbol{CASL(r) = \min(grad(a)) \qquad \forall a \in A}$$

The default security level for an arbitrary rule $\boldsymbol{r}$ is initially **5**. This is based on our assumption that *CrySL* rules are complete, correct and secure by specification. The *CASL* metric restricts the security level only if it specifies an algorithm within Table 4.2 and all other allowed algorithms are considered to be unconditionally secure. Therefore, we assume that a given *CASL* assignment, as ours' in Table 4.2, is correctly verified. Let us demonstrate this with an example for clarification. Assume that we have the given static Algorithm strength graduations below and the CrySL rule $\boldsymbol{r}$ in Figure 3.1. Thus, we determine $\boldsymbol{CASL(r)}$ as follows:

$$\begin{aligned}
&\boldsymbol{A = \{McEliece, RSA, AES, Blowfish\},}\\
&\boldsymbol{grad(RSA) = 4,}\\
&\boldsymbol{grad(McEliece) = 5,}\\
&\boldsymbol{grad(AES) = 4,}\\
&\boldsymbol{grad(Blowfish) = 3,}\\
\Longrightarrow\ &\boldsymbol{CASL(r) = \min(4,5,4,3) = 3}
\end{aligned}$$

Looking up the value **3** at Table 4.1, we can see that this rule has the security level: **CCS (Conditionally Computationally Secure)**.

All Cryptographic Algorithm Security Level (ACASL)

This metric represents the security level of a rule set $\boldsymbol{R}$ for the given cryptographic library. We compute *ACASL* as follows:

$$\boldsymbol{ACASL = \min(CASL(r)) \qquad \forall r \in R}$$

Based on current research and considering existing attacks on cryptographic algorithms, we will assign each algorithm its respective $\boldsymbol{grad}$ value. We choose to do this statically, because if one algorithm becomes weaker in future, one can easily reassign its $\boldsymbol{grad}$ value and reanalyze the specifications. Considering the definitions in Figure 4.4, we assign as

follows.
Through the work by Stevens et al. [SBK+17], we chose to grade the algorithms *SHA-0. SHA-1 and MD5* as weak, since finding collisions of each of them can be done in reasonable time with reasonable resources. We consider algorithms using 64-bit blocks to be weak as well, because of the existing birthday attack. These are: *Blowfish, 3DES, SKIPJACK.*
As mentioned in Section 3.2.1, the *RSA* algorithm can be insecure if chosen with wrong padding schemes or key sizes. Nevertheless there exist secure alternatives, such that we consider *RSA* to be conditionally computationally secure (CCS) [see Section 3.2.1]. The *RC5* algorithm has the same grading as *RSA* Figure 3.4.
For the following algorithms, if chosen with correct key sizes, there are no sophisticated attacks that can break them in reasonable time. Therefore, these are considered to be computationally secure (CS) [BSI20]. The algorithms and protocols fitting this category are: *SHA-2. SHA-3, Elliptic Curve Cryptography (ECC), ElGamal, Diffie-Hellmann, AES and various algorithms based on AES.* Algorithms, we consider to be unconditionally secure (US) [BSI20] are: *FrodoKEM-976, FrodoKEM-1344 and Classic McEliece.*

In Table 4.2, we give a comprehensive overview about the all the assignments.

alg	***grad*(alg)**
MD5, SHA-0, SHA-1	2
DES, 3DES, SKIPJACK, Blowfish	2
SHA-2, SHA-3	4
RSA, RC5	3
Elliptic Curve Cryptography (ECC)	4
Diffie-Hellmann, ElGamal	4
AES, Camellia, Serpent, Twofish, Rijndael, RC6	4
McEliece, FrodoKEM-1344, FrodoKEM-976	5

Table 4.2: Graduation Values for Algorithms

4.2 Prioritization of the metrics

All the previously introduced metrics [see Section 4.1] are dedicated to measure *CrySL* sections [KSA+19]. We presented metrics dedicated for specific *CrySL* sections and one general metric *EPS* that can be applied on all sections. Each of our metrics mainly analyze the properties *Learnability, Memorability and Errors* leading to better or worse *Satisfaction* [HR00] of the analyzed *Crypto API* rule set.

We treat the sections `OBJECTS, ENSURES, NEGATES, REQUIRES` equally, meaning that their weight in rating *secure usability* is equal. Because of this, we only count the number of elements and compute average values over all present sections with *EPS, AEPS* and *AAEPS*. Here, we compare the general quantity of elements between sections and through the metric *AAEPS*, the equal priority of `OBJECTS, ENSURES, NEGATES, REQUIRES` [KSA+19] is ensured.

We have designed *PPM, APPM and AAPPM* dedicated to the `EVENTS` section [KSA+19], which is evaluated separately. Thereby, we assign the `EVENTS` section a higher priority than `OBJECTS, ENSURES, NEGATES` and `REQUIRES`. Using the wrong parameter or less then enough parameters can easily be established through using a faulty overloaded method or constructor. In addition to this, we follow the idea that remembering and learning correct parameters with argument positions is more complex than correct method names. Even though *remembering* has *cognitive complexity level 2* [see Figure 3.3], we assign the `EVENTS` section a higher priority. Having the same priority are the `ORDER` and `FORBIDDEN` sections [KSA+19], but for different reasons. We measure `ORDER` in addition to *EPS*, also with the *Ratio of Accepted Orders (RAO)* [see Section 4.1], which focuses on the property *Errors* [HR00]. Being provided with secure methods in the `EVENTS` section, it requires a further step of applying these in correct order and completeness. *RAO* determines the variety of the usage order that the allowed methods have [see Section 4.1]. We consider this to be, analogously to *PPM*, more complex than the `OBJECTS, ENSURES, NEGATES` and `REQUIRES` sections and therefore, the higher priority assignment. The `FORBIDDEN` section together with the helper method `noCallTo` [KSA+19] are assigned the same priority as `EVENTS` and `ORDER`. With *ANFM* [see Section 4.1] the total number of forbidden methods is determined, which represents the number of methods that should be removed from the library or redesigned securely. Unlike methods that are not specified at all, the usage of forbidden methods are known to be dangerous. Though *CrySL* follows a white listing approach [KSA+19], the forbidden methods can be viewed as a blacklist in *CrySL* and therefore, have a higher priority. The highest priority is assigned to the `CONSTRAINTS` section [KSA+19], since several security aspects are specified there. For instance, one can specify the allowed algorithms, the key sizes or implications with specific conditions [KSA+19]. We measure its complexity by counting the number of elements *(EPS)* and finding the most insecure algorithm that is allowed to use (*CASL*) [see Section 4.1]. In the `CONSTRAINTS` section we can separate *security* from *usability*, since *CASL* does only measure the security of the respective rule and with *ACASL* [see Section 4.1] the library's rule set. Since we are able to measure both *security* and *usability*, this section is assigned the highest priority.

Let us comprehend the priority assignment for specific *CrySL* sections [KSA+19] in the following Table 4.3.

CrySL Section	**Applying Metrics**	**Priority**
`OBJECTS`	*EPS*	0
`ENSURES`	*EPS*	0
`NEGATES`	*EPS*	0
`REQUIRES`	*EPS*	0
`ORDER`	*EPS, RAO*	1
`FORBIDDEN`	*EPS, NFM*	1
`EVENTS`	*EPS, PPM*	1
`CONSTRAINTS`	*EPS, CASL*	2

Table 4.3: CrySL Section *[KSA+19] Priorities*

Note: *A higher priority number indicates a higher priority value.*

Rating The Metrics

Our rating system is based on comparing at least two library rule sets with each other to determine, which library provides better *secure usability.* We mentioned that *security* and *usability* are interconnected and mostly cannot be separated [see Section 3.3.3]. Therefore, each metric contributes equally to build the final result. The priorities are realized through evaluating all metrics and weighing each of them equally for the final result. For the case of the *ORDER* section [KSA+19] with priority **1** [see Table 4.3], for instance, we measure *AAEPS* and *ARAO*, where latter is dedicated only to that section. The `OBJECTS` section, as a different example with priority **0** [see Table 4.3], has only one applicable metric, which is *EPS.* This way we consider the priorities for each section in the analysis. To rate the rules sets of libraries, we only consider metrics applicable on the whole set, as listed in Table 4.4.

4.3 Automated Rating

Our automated rating is based on the proposed metrics [see Section 4.1] and their prioritization [see Section 4.2]. Let us demonstrate this with the example in Table 4.4, where A and B are library rule sets.

Metric	**A**	**B**	**Winner**
$AAPPM$	1	2	A
$AEPR$	1	2	A
$AAEPS$	2	1	B
$AAEPS_{Constraints}$	2	1	B
$ANFM$	1	2	A
$ARAO$	30%	50%	B
$ACASL$	3	5	B

Table 4.4: Example Results

A has **3** wins and B has **4** wins in this example. With a difference of **1**, ***B*** is rated to be better *secure usable* than ***A***. Since this a comparative rating system, one can see that the rating requires at least two library rule sets.

CHAPTER

5 EVALUATION

The goal of this chapter is to investigate on our third and fourth research questions. To recall them, these were targeted on the application and validity on our developed metrics [see Section 4.1] to draw a comparable and final result at the end. This chapter is structured as following. We first introduce our working environment to perform our analysis and evaluation of the given library rule sets [see Section 5.2]. Then, we apply our metrics on the rules, show and compare the results we get from the analysis to determine which library has the best *secure usability*. Based on the obtained results, we will validate the existence of our metrics and show their right of existence.

5.1 Setting up the Environment

As mentioned in previous chapters, we use the Java programming language as our operating language. Since the static analysis tool CogniCrypt [KNR+17], which uses the *CrySL* language [KSA+19] for its analysis, is an Eclipse Plugin, we have chosen Eclipse as our *Integrated Development Environment (IDE)* as well. Note, that for the development of the metrics we only require the correct *Java* environment. Therefore, there is no urgency to use an *IDE* at all. More details are presented in the following.

- **Eclipse Version:** 2019-06 (4.12.0)
- **Eclipse Build id:** 20190614-1200
- **Operating System:** Windows 10, 64-bit
- **Java version:** 1.8.0_251

5.2 External Resources

The *CrySL* rules we used for our analysis are for the *Google Tink* [goo], *BouncyCastle* [bou] and *JCA* [jca] libraries. We obtained them from the public github repository *Crypto-API-Rules* by the *Collaborative Research Center CROSSING at TU Darmstadt* [gita]. To parse the *CrySL* rules, we used existing Java classes of the public *Crypto-Analysis* repository [gitb]. More specifically, we extended the `CrySLModelReader` class to obtain operable information presented as `CrySLRule` object.

5.3 Structure

To perform our analysis, we implemented helper tools and classes in the Java programming language [gitc]. For the evaluation of our non averaging metrics [see Section 4.1], we designed the `MetricsGenerator` class that takes a *CrySLRule* object [gitb] as argument and creates all metrics dedicated to a single *CrySLRule* [see Section 4.1] by instantiation. The `AverageMetricGenerator` generates all metrics that compute values over the whole rule set, such as computing averages or determining the security level. The respective results are extracted into a `.txt` file to present as graphs later. At the end we call the `ResultGenerator` class to determine, which rule set or sets are rated to be the best *secure usable*, depending on the results of the `AverageMetricGenerator` results.

5.4 Application and Results

In the following, we apply our metrics on the libraries' rules [see Section 5.2], show and compare the results. We assume that the rule specifications for each library are complete, secure and correct, meaning that with usage of these rules the corresponding *API* is considered to be secure usable. Independent of our metrics [see Section 4.1], we follow the principle that the fewer rules a library requires, the better is its *secure usability* by release. Note that, comparing the rule sets with each other may not be fair, since our assumption that the sets are complete, may not be true. Moreover, the rules for respective libraries can support different cryptographic features. Nevertheless, providing average values and applying quantitative metrics for the rule sets circumvents this problem to a certain extent, such that we obtain comparable and representative values respectively.

The *JCA* library contains **46**, *BouncyCastle* **31** and *Google Tink* **24** *CrySL* rule specifications. Here, we can see that the *Google Tink* library requires the fewest rules to guarantee *secure usability.* Without considering the content of the rules, this indicates that the *Google Tink* library is likely to be more secure than the other two.
For figures in this chapter, one should know that the class index $\boldsymbol{i}$ refers to the $\boldsymbol{i}$-th *CrySL* rule in the respective library rule set. Depending on the applied metric, the *Amount* axis of a figure represents the relative number of elements for respective metrics, such as *Elements per Rule (EPR)* or *Parameters per Method (PPM).* The rules at *class index*

i do not implement similar features, meaning that, for instance, the element at index **7** might be a `KeyGenerator` rule for one library and a `Mac` rule for another.
We first measure the overall size and complexity of the rules in a library rule set with the *Elements per Rule (EPR)* metric. The results are illustrated in Figure 5.1.

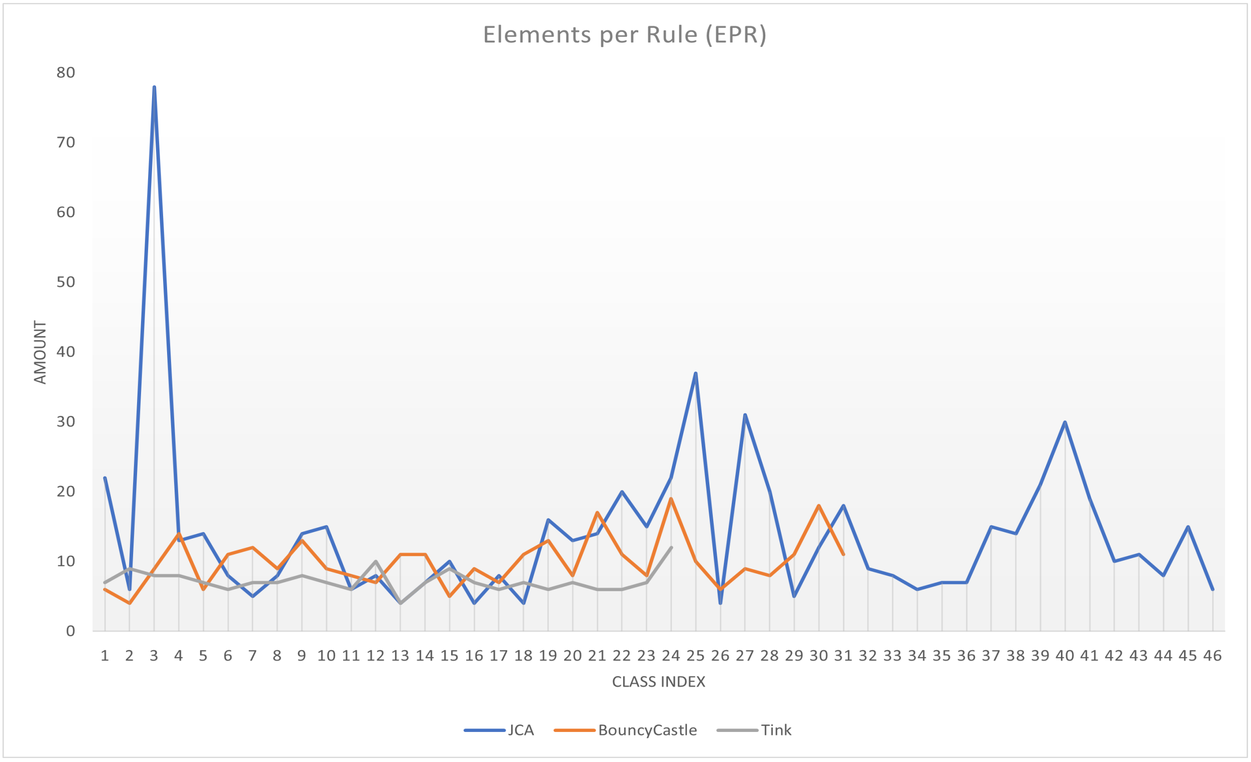


Figure 5.1: *Elements per Rule (EPR)* metric on libraries

To demonstrate an example, consider third rule of the libraries (class index **3**). Here, we can see that *JCA* contains **78**, *BouncyCastle* **8**, *Google Tink* also **8** elements. Through measuring the *EPR* of the libraries [see Figure 5.1], we know that *Google Tink* has the fewest elements per rule, followed by *BouncyCastle* and *JCA* in that order. This result supports our intuition of *Google Tink* being the most secure usable library under the given three. The *EPR* values of *BouncyCastle* are not much higher than *Google Tinks'*. *JCA* on the other hand, has significantly higher *EPR* values than the previous libraries, such that it indicates worse *secure usability*.

Moving forward to the next metric, we evaluate the complexity of methods in rules specified in the `EVENTS` section [KSA+19]. Here, we compute an average on the amount of parameters per method in a rule (*APPM*) [see Section 4.1] to analyze their *Memorability* and *Learnability* [HR00]. The results are shown in Figure 5.2 below.

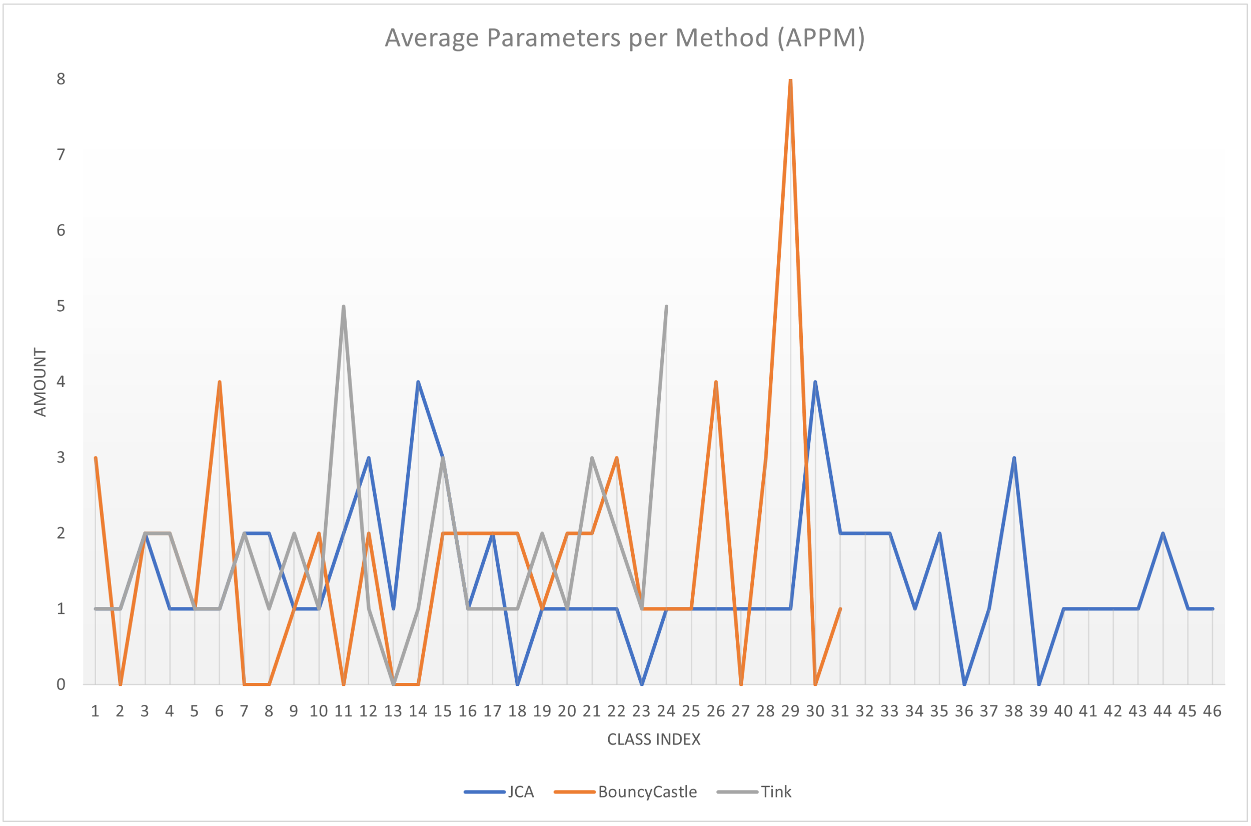


Figure 5.2: *Average Parameters per Method (APPM)* metric on libraries

As an example, the average method for the third rule in each rule set (class index **3**) has **2** parameters per method. Through comparing the *APPM* values for each library, it can be observed that the values fluctuate for each class index ***i***, such that there is no unique argument on the *secure usability* of any given library. While the *APPM* values of *Google Tink and JCA*, ranging between **0** and **5**, do not show much divergence, the *APPM* of *BouncyCastle* has high variation from one to the next rule. To serve an example, in the rules from class index **1** to **15**, the *APPM* values of *BouncyCastle* fluctuate between being double the values of *Google Tink and JCA* for one rule, and half the value in the next rule. To compare the secure usability between these libraries, the result of this measurement is not conclusive and should not be interpreted on its own in this case.

Next, we measure the *Ratio of Accepted Orders (RAO)* [see Section 4.1]. Here, the goal is to find out the percentage of accepted usage patterns under the correctly given usage order and methods in the *ORDERS* section [KSA+19]. The results are shown in Figure 5.3.

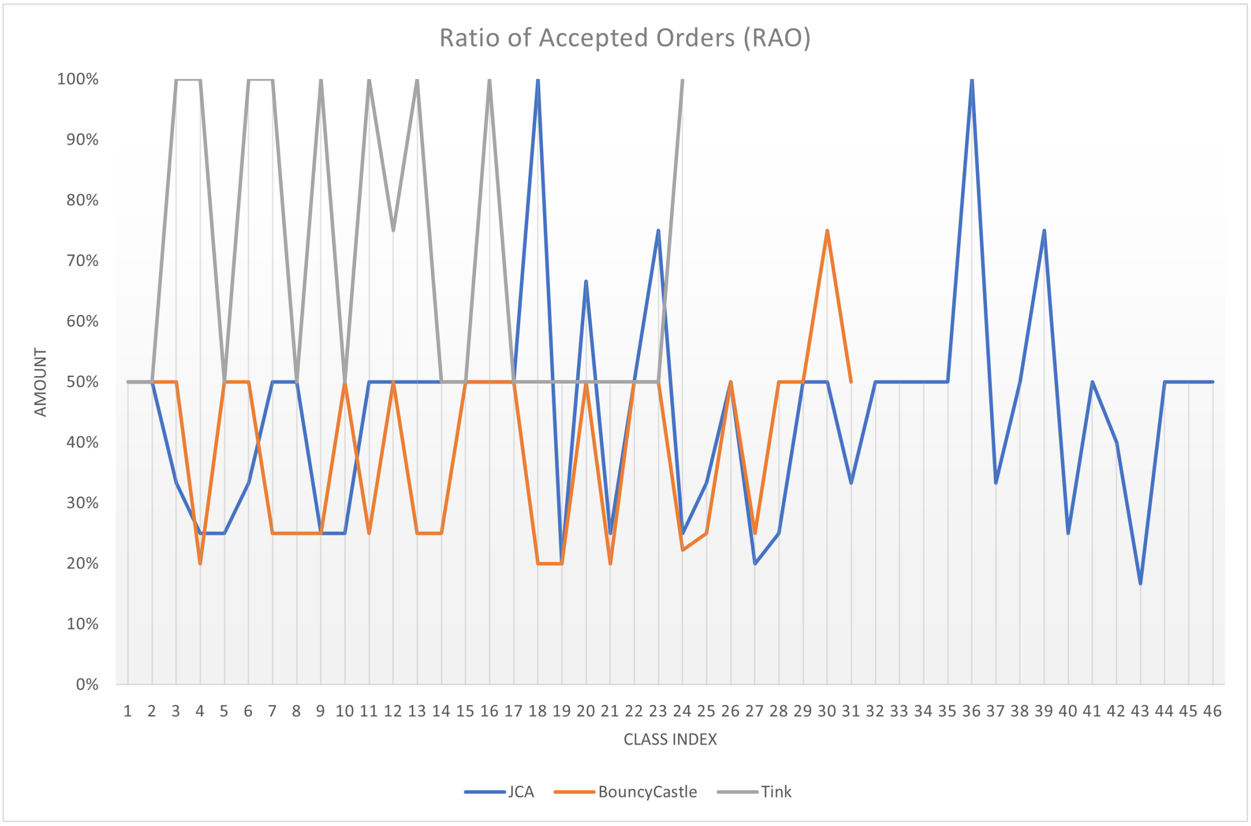


Figure 5.3: *Ratio of Accepted Orders (RAO)* metric on libraries

Unlike the results for the *APPM* measurement, this metric (*RAO*) has more decisive results. Having the highest *RAO* values, *Google Tink* offers more implementation variety and a lesser error probability [see Section 4.1] than the other two libraries. All rules for the *Google Tink* library are above **50%**, indicating that the `ORDER` specifications for implementation are more flexible and provide more accepting patterns [see Section 4.1].

We continue with analyzing the security level of each library. For this, we measure the *cryptographic algorithm security level (CASL)* [see Section 4.1] and take the respective minimum *CASL* in the rule set. Since we are interested in the security level of the whole library, we calculate the *All Cryptographic Algorithm Security Level (ACASL)* [see Section 4.1] as well. Let us recall the grading of *CASL* [see Table 4.1]. The security levels range from **1** to **5**, where a higher number indicates better security [see Table 4.1]. The results are illustrated below.

JCA	**BouncyCastle**	**Google Tink**
2	5	5

Table 5.1: *All Cryptographic Algorithm Security Level (ACASL)* for libraries

The application of *CASL* showed that *Google Tink's*, *BouncyCastle's* and **93.48%** of *JCA's Cryptographic Algorithm Security Level* [see Section 4.1] of rules either do not specify any algorithm restrictions in the `CONSTRAINTS` section [KSA+19] or the specified algorithms are considered to be *unconditionally secure (US)* [see Table 4.1]. Hence, *BouncyCastle* and *Google Tink* both are assigned the maximum security level. *JCA* contains two rules allowing to use algorithms that are *weak (W)* and one rule that is *conditionally computationally secure (CCS)* [see Table 4.1].
As the evaluation of *ACASL* shows [see Table 5.1], the rule set of the *JCA* library has a *weak (W)* security level, which is a concern for each developer using *JCA* for cryptographic functionalities. Both *BouncyCastle* and *Google Tink* use cryptographic algorithms that we assigned to have the maximum security level and therefore, can be equally treated in this category.

Next, we apply metrics that calculate the average values of the previous metrics. Analyzing the average values is very valuable for our evaluation, since it ignores the total amount of rules between the sets. The goal is to compare the average rule as a representative for each library's rule set. Let us recall the relevant metrics [see Section 4.1] in the following:

> *AAPPM*: Measures the average number of parameters per method of the average rule in a library
>
> *AEPR*: Measures the average number of elements of the average rule in a library
>
> *AAEPS*: Measures the average sections of the average rule in a library
>
> $AAEPS_{Constraints}$: Measures the average elements per `CONSTRAINTS` section [KSA+19] of the average rule in a library
>
> *ANFM*: Measures the total number of the forbidden methods in a library
>
> *ARAO*: Measures the average ratio of accepted orders of the average rule in a library
>
> *ACASL*: Measures the minimum cryptographic algorithm security level of the library

The measurement results of the above metrics are presented in Table 5.2.

Metric	**JCA**	**BouncyCastle**	**Google Tink**
AAPPM	1.39	1.68	1.71
AEPR	14.07	10.03	7.25
AAEPS	2.59	1.65	1.29
$AAEPS_{Constraints}$	2.78	1.55	1.04
ANFM	17	0	0
ARAO	45.22%	39.59%	69.79 %
ACASL	2	5	5

Table 5.2: Average Metric Results [see Section 4.1]

We found out previously that the comparison of the *APPM* values, concerning the given library rule sets, is not decisive and should not be judged on its own. The measurement of (*AAPPM*) confirms this statement, since they show no significant divergence between each value. Even though there is not a huge difference, our automated rating system treats *JCA* as the winner of this measurement *(AAPPM)* due to small edge [see Section 5.6].
The *AEPR* value of *Google Tink* is half the value of *JCA's* and also significantly smaller than *BouncyCastle's*. Therefore, *Google Tink* offers the best *secure usability* in this category. The measurement of *AAEPS* provides almost equal results relatively, such that *Google Tink* again is the better choice for *secure usability.* An interesting point we recognized by relating the results of *AAEPS* to *AEPR* is, that the elements of the average rule are evenly distributed over the *CrySL sections* [KSA+19] for all library rule sets. As an example, for *JCA* we calculate $\frac{\mathbf{14.07}}{\mathbf{2.59}} \approx \mathbf{5}$. This indicates that the average rule in *JCA's* rule set has about **5** sections. Similarly *BouncyCastle* has **6** sections and *Google Tink* **5** sections.
The inspection of the *ARAO* values shows that, although *BouncyCastle* has smaller *AAPPM, AEPR* and *AAEPS* values than *JCA*, it specifies a more restricted usage order for the methods being used than *JCA*. This is due to a smaller *ARAO* value (percentage). Also here, *Google Tink* has the best *secure usability*, since it has the highest *ARAO* value and conclusively, provides more flexibility and less memorization of the usage order.
We recall that forbidden methods consist of elements specified in the `FORBIDDEN` section and methods forbidden by the helper function `noCallTo` [KSA+19]. Comparing *ANFM*, we observe that *Google Tink* and *BouncyCastle* do not forbid any methods, but *JCA* on the other side forbids **17**, making *JCA* clearly the worst in this measurement. The average elements for the `CONSTRAINTS` section [KSA+19] again indicates that *Google Tink* has the best *secure usability* followed by *BouncyCastle* and *JCA* in that order. Finally,

through comparing *ACASL*, other than *JCA*, which is rated as weak, both of the other rule sets are rated as unconditionally secure (US) [see Table 4.1]. The comparison of the previous results indicates that *Google Tink* provides the best secure usability, followed by *BouncyCastle* and *JCA*.

5.5 Validating the Metrics

Our metrics [see Section 4.1] measure various quantitative and qualitative aspects of *CrySL* rules [KSA+19]. As presented in Table 5.2, their measurements result in numeric values, providing good comparability. Our principle of counting elements assigned to a *CrySL* rule or *CrySL* section [KSA+19] is analogous to calculating the *Attribute Complexity (AC)* by Misra et al. [MAFSD18]. *AC* calculates the complexity of data members assigned to a class [see Section 3.6]. Analogously, measuring the elements of a section *(EPS)*, the elements of a rule *(EPR, NFM)* or the parameters of a method *(PPM)* can be treated as data members as well. Hence, we adapted and extended *AC* for *CrySL* rules. Our *Ratio of Accepted Orders (RAO)* [see Section 4.1] measures the variety that the secure methods, specified in the `EVENTS` section, can be used. It is intuitive that the more usage variety is offered, the less the user is forced to remember certain patterns. Assuming that the *CrySL* rule would not exist, but the user knew which methods were secure, he or she still could cause security threats due to less usage variety, such as the correct implementation order. Therefore, *RAO* is a good representative for the variety of implementation possibilities and provides good comparability as shown in the previous section.
Our *Cryptographic Algorithm Security Level (CASL)* implements the suggestions of Jorstad et al. [JL97]. Determining the weakest security level within a library rule set to be the total security level of that library (ACASL [see Section 4.1]) follows the worst case scenario of a developer using the most insecure algorithm that is allowed for implementation. Due to our numeric graduation of algorithms [see Table 4.1, Table 4.2], a comparability is provided and applied successfully in the previous section.

5.6 Final Rating

For the final rating, we perform the automated analysis. The obtained results do correspond to the results in the previous section and are presented in Table 5.3.

Metric	**JCA**	**BouncyCastle**	**Google Tink**
$AAPPM$	x		
$AEPR$			x
$AAEPS$			x
$AAEPS_{Constraints}$			x
$ANFM$		x	x
$ARAO$			x
$ACASL$		x	x

Table 5.3: Winner Determination [see Section 4.1]

An ***x*** in Table 5.3 marks the best value (winner) for that measurement. So finally, we consider the library *Google Tink* with **6** wins to provide the best *secure usability* under these three libraries. It is followed by *BouncyCastle* with **2** wins followed by *JCA* with **1** win.

CHAPTER

6 CONCLUSION

The development of secure software these days require modern cryptographic functionalities, such as password hashing or file encryption. Hence, the requirement for secure Cryptographic Application Prorgramming Interfaces (Crypto APIs) increases as well. We defined and developed the concept of a cryptographic library to be *secure usable.* It comprises easy to use *Crypto APIs*, meaning the usage is not too cognitively challenging for the developers. Simultaneously, it must provide secure cryptographic algorithms and should forbid the usage of weakly specified algorithms.
This work aimed to investigate on rating the *secure usability* of *Crypto APIs* based on *CrySL* rule specifications [KSA$^+$19]. We developed a set of metrics [see Section 4.1] to measure which of the given cryptographic libraries (*BouncyCastle, Google Tink and JCA*) have the best *secure usability.* We implemented and performed these metrics in the *Java* language, which can be replicated or extended for further investigations [gitc]. Our evaluation in Chapter 5 showed that the *Google Tink* library [goo] has the best secure usability followed by *BouncyCastle* [bou] and *JCA* [jca]. Even though, we got a winner for being the most *secure usable*, our result still has its limitations, which are addressed in Section 6.2. Since the library rule sets are in continuous development, we emphasize the need to redo the evaluation as performed in Chapter 5 in future to stay current with the results.

6.1 Answering Research Questions

For the first research question our goal was to find out what properties of *CrySL* specifications were relevant to assess the *secure usability* of *Crypto APIs.* Following the semantics of our metrics [see Section 4.1] the main properties that we used to rate and compare the metrics are *Memorability, Learnability, Errors* and *Satisfaction* [see Section 3.3.3] of *CrySL* specifications. Depending on the applied metric, for the rating we put more emphasis on specific *API* properties. To measure, for instance, the quantity of elements in a rule with the *Elements per Rule (EPR)* metric [see Section 4.1], the properties we considered were *Memorability* and *Learnability* [HR00]. For *EPR's* assessment, we followed the principle that the fewer elements a rule has, the better is its *Memorability* and *Learnability* and so is its *secure usbaility.* The other metrics were rated with similar concepts.

For the second research question we extended and adapted existing metrics, such as *Attribute Complexity (AC)* by Misra et al. [MAFSD18] and measuring cryptographic algorihtm strength by Jorstad et al. [JL97]. Through introducing metrics that measure various quantitative aspects of *CrySL* rules [KSA+19], such as *Parameters per Method (PPM) or Elements per Rule (EPR)* [see Section 4.1], we successfully adapted the *Attribute Complexity* metric [MAFSD18] for *CrySL* rules. Following the suggestions by Jorstad et al. [JL97] of measuring cryptographic algorithm strength, we developed the *CASL* metric [see Section 4.1], which determines the security level of a *CrySL* rule. Furthermore, we computed average values over the whole rule set for our automated rating. We build this rating system in consideration of the prioritization for each *CrySL* section [see Section 4.2]. Thus, we guaranteed the different influences of each *CrySL* section [KSA+19] on the final rating based on their priorities [see Table 4.3].

For the third research question we validated the right of existence for our introduced metrics in Section 4.1 and Section 5.5. Based on existing metrics, measuring attributes of classes [MAFSD18], we made analogous measurements for *CrySL* elements [KSA+19] by adapting and extending these for *CrySL* rules. Here, we focused on *CrySL* elements of respective sections and rules [see Section 4.1]. Furthermore, to determine the security level of *CrySL* rule specifications, we used the graduation system introduced by Jorstad et al. [JL97]. Considering that a user is allowed to use the least secure algorithm in the specification, we followed a lower bound principle and took the minimum security level as representative for the library's security level.

The fourth and final research question was to interpret the results and show, which of the given libraries are best for secure software development. Our results in Chapter 5 show that *Google Tink* [goo] is best, followed by *BouncyCastle* [bou] and *JCA* [jca]. Note, that the result has its limitations, which are addressed in Section 6.2.

6.2 Limitations

We based our metrics on measuring largely cognitive aspects of *CrySL* rules. We also measured one security aspect [see Section 4.1]. Even though for secure software development *security* and *usability* coincide, we still miss metrics to measure important cryptographic properties, such as key size, modes of operation and algorithm efficiency. Having also metrics to measure these aspects, it certainly would contribute to better security assessment of the libraries.

For the analysis and evaluation, we assumed that the *CrySL* rules for the libraries are complete, correct and secure. Considering that the rules for each library are still in development, we know that the completeness part of our assumption is not necessarily fulfilled. Hence, for our automated rating we only considered metrics computing average values [see Section 5.6], allowing better comparability. Overlaying the missing completeness aspect with comparing the average rules fairly cannot have the same precision as evaluating a complete rule set. Performing the same analysis with a more complete rule set may lead to different results as we obtained.

A further aspect we missed for our analysis is a user study on *Memorability, Learnability, Errors and Satisfaction* [see Section 3.3.3] specifically for *CrySL* rules. Having this would introduce the possibility to develop a generic rating for any library without the requirement of more library rules (see Section 6.3 for more details). In contrast to that, in our analysis we rely on at least two rule sets and determine, which of the given sets has better *secure usability.*

6.3 Future Work

Our rating tool [gitc] can easily be extended in order to measure more aspects of *CrySL* rules. Hence, the foundation for further research is provides with this work. As suggested in Section 6.2, there is a lack of metrics measuring the cryptographic security aspect. Adding these would improve the final rating result or might change it. Since, the *API* rules [gita] are in continuous development, it is recommended to rate and compare the *secure usability* from time to time again to be synchronized with current rule sets.

Conducting a user study on cognitive complexity aspects, such as *Memorability, Learnability, Errors and Satisfaction* [see Section 3.3.3] would introduce the possibility to develop a generic rating on the *secure usability* of *CrySL* rules. Having this, the requirement of having at least to rule sets for comparison would not be necessary and we could rate a rule set based on the results and conclusions from the user study. Therefore, the conduction of a user study for *CrySL* rules could be part of future work. A possible study, that we suggest, could be to make a usability survey on the introduced metrics in this work [see Section 4.1]. Thus, a grading system for each presented metric could be developed.

The current metrics analyze only *CrySL* rules. The comparison of the rule sets to all actual components of the library can be beneficial. To demonstrate this on a realistic example, let us assume that a library (not rule set) provides the three different options A,B and C to encrypt a file. Furthermore, assume that the *CrySL* rules only allow option B to be implemented, meaning only B is considered as *secure usable.* The ratio of secure implementations for file encryption, provided by the library itself, would be **1 : 3** in this example. Integrating the concepts of this idea, would introduce a new perspective for rating the *secure usability* of cryptographic libraries.

BIBLIOGRAPHY

[ABF+17] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. Comparing the usability of cryptographic apis. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 154–171. IEEE, 2017.

[And93] Ross Anderson. Why cryptosystems fail. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 215–227, 1993.

[AZZ+10] Hamdan Alanazi, B Bahaa Zaidan, A Alaa Zaidan, Hamid A Jalab, M Shabbir, Yahya Al-Nabhani, et al. New comparative study between des, 3des and aes within nine factors. *arXiv preprint arXiv:1003.4085*, 2010.

[BL16] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-) security of 64-bit block ciphers: Collision attacks on http over tls and openvpn. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 456–467, 2016.

[Blo06] Joshua Bloch. 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*, pages 506–507, 2006.

[Böc11] Johannes Böck. *RSA-PSS–Provably secure RSA Signatures and their Implementation v1. 0.1.* PhD thesis, Diplomarbeit, Humboldt-Universität zu Berlin, 2011. http://rsapss. hboeck . . . , 2011.

[bou] Bouncy Castle Crypto API. `https://www.bouncycastle.org/`.

[BS15] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *Draft 0.2*, 2015.

[BSI20] BSI – Technical Guideline - TR-02102-1 - Cryptographic Mechanisms: Recommendations and Key Lengths, 24th March, 2020.

[BSY18] Hanno Böck, Juraj Somorovsky, and Craig Young. Return of bleichenbacher's oracle threat ({ROBOT}). In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 817–849, 2018.

[Cop01] Don Coppersmith. Finding small solutions to small degree polynomials. In *International Cryptography and Lattices Conference*, pages 20–31. Springer, 2001.

[DJ06] Tom St. Denis and Simon Johnson. *Cryptography for Developers*. Syngress, 2006.

[Dou08] Andre Doucette. On api usability: An analysis and an evaluation tool. *CMPT816-Software Engineering, Saskatoon, Saskatchewan, Canada*, 2008.

[dSB09] Cleidson RB de Souza and David LM Bentolila. Automatic evaluation of api usability using complexity metrics and visualizations. In *2009 31st International Conference on Software Engineering-Companion Volume*, pages 299–302. IEEE, 2009.

[gita] Crypto-API-Rules. `https://github.com/CROSSINGTUD/Crypto-API-Rules`.

[gitb] CryptoAnalysis. `https://github.com/CROSSINGTUD/CryptoAnalysis`.

[gitc] Thesis-Analysis. `https://github.com/furky97/Thesis/tree/dev`.

[goo] Google Tink. `https://opensource.google/projects/tink`.

[GS16] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[HR00] David M Hilbert and David F Redmiles. Extracting usability information from user interface events. *ACM Computing Surveys (CSUR)*, 32(4):384–421, 2000.

[jca] Java Cryptography Architecture. `https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html`.

[JL97] Norman D Jorstad and TS Landgrave. Cryptographic algorithm metrics. In *20th National Information Systems Security Conference*, pages 1–38, 1997.

[Ker] Auguste Kerkhoffs. La cryptographie militaire. `https://www.petitcolas.net/kerckhoffs/index.html`, accessed 13th August 2020.

[KL14] J. Katz and Y. Lindell. *Introduction to Modern Cryptography, Second Edition.* Chapman & Hall/CRC Cryptography and Network Security Series. Taylor & Francis, 2014.

[KNR+17] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. Cognicrypt: supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–936. IEEE, 2017.

[KSA+19] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, 2019.

[LCWZ14] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail? a case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, pages 1–7, 2014.

[MA08] Sanjay Misra and Ibrahim Akman. Weighted class complexity: a measure of complexity for object oriented system. *Journal of Information Science and Engineering*, 24:1689–1708, 2008.

[MAFSD18] Sanjay Misra, Adewole Adewumi, Luis Fernandez-Sanz, and Robertas Damasevicius. A suite of object oriented cognitive complexity metrics. *IEEE Access*, 6:8782–8796, 2018.

[SBK+17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. In *Annual International Cryptology Conference*, pages 570–596. Springer, 2017.

[SK15] Thomas Scheller and Eva Kühn. Automated measurement of api usability: The api concepts framework. *Information and Software Technology*, 61:145–162, 2015.

[SKW+00] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. The twofish encryption algorithm. 09 2000.

[SW03] Jingqiu Shao and Yingxu Wang. A new measure of software complexity based on cognitive weights. *Canadian Journal of Electrical and Computer Engineering*, 28(2):69–74, 2003.

[VP17] R Vasantha and R Satya Prasad. An advanced security analysis by using blowfish algorithm. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 2(5):1031–1036, 2017.

[Wan03] Yingxu Wang. On cognitive informatics. *Brain and Mind*, 4(2):151–167, 2003.

[Wey88] Elaine J Weyuker. Evaluating software complexity measures. *IEEE transactions on Software Engineering*, 14(9):1357–1365, 1988.

SOFTWARE VERSIONS

Resource	**Commit**
https://github.com/CROSSINGTUD/Crypto-API-Rules	`11c96359c`
https://github.com/furky97/Thesis	`e4385a9a1`

Table 6.1: Software Versions

LIST OF FIGURES

LIST OF TABLES