MuBench-JCE — A Misuse-Detection Benchmark for the Java Cryptography Extensions API

MuBench-JCE — Ein Benchmark von fehlerhaften Verwendungen der Java Cryptography Extensions APIs

Mattis Manfred Kämmerer (B.Sc. Informatik)

Technische Universität Darmstadt
Department of Computer Science
Software Technology Group

Examiner: Prof. Dr.-Ing. Mira Mezini
Supervisor: Sven Amann, M.Sc.

Submission Date: 10.05.2017

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Abstract

English

Recent studies on the usage of cryptographic Application Programming Interfaces (APIs) show that developers often use them in insecure ways [EBFK13, NKMB16]. Among the reasons for such problems are a lack of knowledge about cryptographic concepts, deficiencies in the design of the cryptographic APIs, and missing support for high-level concepts in cryptography libraries. This leads to a prevalence of vulnerabilities of today's software.

In the past, API-misuse detection has been proposed as a means to automatically detect problems in the usage of APIs [ANN+16]. While this approach is generally promising, we ask ourselves whether it is capable of finding problems specific to cryptographic APIs. In this thesis, we want to systematically approach this question on the example of *Cipher*, the encryption and decryption API, which is part of the Java Cryptography Extensions (JCE). To this end,

- We use Boa [DNRN13] to mine GitHub for projects using *Cipher*. We manually review 26 projects, and investigate, why *Cipher* is misused as frequently.
- We assess whether the Misuse Classification (`MuC`) covers all types of misuses, i.e., violations of usage directives, by comparing it to the general API-usage directives reported by Monperrus et al. [METM12], and analyze which directives apply for the *Cipher* misuses that we find in our dataset.
- We extend `MuBench`, a benchmark dataset and pipeline for API-misuse detectors, using our *Cipher*-misuse dataset.
- We run four existing API-misuse detectors on the new `MuBench` dataset and assess whether they catch misuses of the JCE. Subsequently, we analyze the capabilities and deficiencies of the detectors in this regard.

After this thesis, we know whether and how *Cipher* differs from general Java APIs, with respect to usage directives and misuses. Also, we know more about the problems of *Cipher* specifically, have a benchmark dataset for `MuBench`, and preliminary results on how existing detectors perform on this benchmark.

Deutsch

Kürzliche Studien über die Benutzung von kryptographischen Application Programming Interfaces (APIs) zeigen, dass Entwickler sie oft auf unsichere Art verwenden [EBFK13, NKMB16]. Unter den Gründen für solche Probleme sind ein Mangel an Wissen über kryptographische Konzepte, Defizite im Design von kryptographischen APIs und fehlende Unterstützung von High-Level Konzepten in kryptographischen Libraries. Dies führt zu einer weiten Verbreitung von Schwachstellen in der heutigen Software.

In der Vergangenheit wurde API-misuse detection als ein Mittel zur automatischen Feststellung von Problemen in der Benutzung von APIs vorgeschlagen [ANN+16]. Obwohl dieser Ansatz generell vielversprechend ist, fragen wir uns, ob es ihm möglich ist, Probleme zu finden, die speziell in kryptographischen APIs auftreten. In dieser Abschlussarbeit wollen wir dieser Frage am Beispiel von *Cipher*, der Verschlüsseungs- und Entschlüsselungs-API, welche Teil der Java Cryptography Extensions (JCE) ist, systematisch nachgehen. Um dies zu erreichen,

- verwenden wir Boa [DNRN13], um auf GitHub nach Projekten, die *Cipher* verwenden, zu suchen. Wir prüfen 26 Projekte manuell und untersuchen, warum *Cipher* so häufig falsch verwendet wird.
- beurteilen wir, ob die Misuse Classification (`MuC`) alle Arten von Fehlern, das heißt Verletzungen von Benutzungsdirektiven, abdeckt, indem wir sie mit den generellen API-Benutzungsdirektiven, von denen Monperrus et al. [METM12] berichten, vergleichen und analysieren, welche Direktive auf die *Cipher* Fehlverwendungen, die wir in unserem Datensatz finden, anwendbar sind.
- erweitern wir `MuBench`, einen Benchmark Datensatz und eine Benchmark Pipeline für API-misuse Detektoren, unter Verwendung unseres *Cipher*-misuse Datensatzes.
- führen wir vier existierende API-Misuse Detektoren auf dem neuen `MuBench` Datensatz aus und bewerten, ob sie *Cipher*-Fehlverwendungen finden. Anschließend analysieren wir die Fähigkeiten und Defizite der Detektoren in dieser Hinsicht.

Nach dieser Abschlussarbeit wissen wir, ob und wie sich *Cipher* von generellen Java APIs in Bezug auf Benutzungsdirektive und Fehlverwendungen unterscheidet. Außerdem wissen wir mehr über die speziellen Probleme von *Cipher*, haben einen neuen Benchmark Datensatz für `MuBench` und vorläufige Ergebnisse, was existierende Detektoren auf diesem Benchmark leisten.

Table of Contents

1 Introduction

The *Java Cryptography Extension* (JCE) is Java's Application Programming Interface (API) for cryptography tasks. In this thesis we focus on the encryption and decryption API. This API hinges mainly on the *javax.crypto.Cipher* class. *Cipher* encapsulates all encryption algorithms and supports third party providers. Its interface is based on the provider pattern for two main reasons: the algorithms vary a lot in their implementation details, and the interface was built to be very backwards compatible—having been like this since its addition to the standard Java platform *J2SE version 1.4* published on February 6th, 2002.

Cipher's interface accommodates the algorithm-specific implementation details by several method overloads, where certain alternatives should only be used with specific types of algorithms. Thus, we suspect it can easily be misused. Due to its critical purpose, bad usage of *javax.crypto.Cipher* likely leads to unnoticed security vulnerabilities. Indeed, previous research [EBFK13, Zie15, NKMB16] shows that the JCE is often used incorrectly. In Chapter 2, we explain *Cipher*'s API in more detail, and discuss why it is commonly misused.

This thesis examines these issues and suggests what could or should be done to help minimize Cipher misuses. We focus on API-misuse detectors, as they are automated tools to support correct usage of general APIs. Misuse detectors classify code snippets, and rank them according to the estimated likelihood of incorrectness. Most misuse detectors rely on pattern mining to build an internal model for correct API usages, and compare the model built with the code to be classified. We show how API-misuse detection is commonly approached, and how it can help developers in Chapter 3.

To evaluate API-misuse detectors on *Cipher*, we conduct an experiment examining whether they can find the common *Cipher* misuses. Using MuBench [ANN+16], we compare some of these detectors directly. MuBench is a benchmarking tool for API-misuse detectors, which currently supports four API-misuse detectors. Currently, only four misuses in the MuBench dataset are *Cipher*-specific. Hence, we need to extend the MuBench data to gain meaningful results for this particular API.

There are existing datasets from previous research [Zie15], representing general JCE usage. However, these contain only little detailed information about each individual data point and are derived from surveys and disassembled applications. Alas, they do not suffice for the MuBench benchmarking process, as they miss the original source code. We instead use Boa [DNRN13] to create a new dataset of *Cipher* misuses. Boa provides snapshots of GitHub and Sourceforge and allows the user to traverse these snapshots using a visitor pattern. The underlying tree structure contains nodes for repositories, revisions, changed files, etc., but notably also *Abstract Syntax Trees*. Using this feature, we retrieve a list of GitHub URLs to files using *javax.crypto.Cipher*—a sample of which we manually review, using a specification created by Krüger [Krü]. This mining and review process is described in Chapter 4.

Using the review results from mining *javax.crypto.Cipher* usages, we add most of the found misuses to the MuBench dataset. As some detectors require compiled source code, we can only add the review results from compilable projects. This gives us a set of 15 misuses, spread over ten projects.

With the new JCE dataset in MuBench, we run the benchmark and evaluate the results in Chapter 5. Our findings show that the four API-misuse detection approaches, `DMMC` [MM13], `GROUMiner` [NNP+09], `Jadet` [WZL07], and `Tikanga` [WZ11], do not work on *Cipher*. Finally, we suggest what future work can do to improve misuse detection for APIs such as *Cipher*.

2 Cipher

The *Java Cryptography Extension (JCE)* is part of the *Java Platform, Standard Edition* since the *Java Development Kit (JDK)* version 1.4, which was published on February 6th, 2002. It is an application programming interface (API) which handles cryptographic tasks, such as encryption, authentication, and key management.

In this thesis, we focus on *Cipher*. Cipher is JCE's interface for all encryption and decryption. It is intended to be used as an interface for all encryption algorithms, and to this end, its architecture is based on the provider pattern, which supports the use of third party providers to provide the algorithms. The next section, Section 2.1, gives an overview of how the *Cipher* API should be used. Then, we introduce the specification we use in our *Cipher* usage review in Section 2.2. Lastly, we discuss some possible problems of *Cipher* in Section 2.3.

2.1 Using *Cipher*

In this section, we give an introduction to how *Cipher* can be used, and what effects the provider architecture has on its usability.

In Figure 2.1, we show an example usage of *Cipher*. In the example, we provide methods for encryption and decryption in *AES* using *Cipher Block Chaining Mode* (CBC) with *PKCS7* padding. We generate a 256 bit key for the encryption. First, we create a *KeyGenerator* for "*AES*". We initialize the generator to generate 256 bit keys in line 11 and generate a new key in line 12, using *KeyGenerator.generateKey()*.

Since we want to use CBC mode, we need to generate an Initialization Vector (IV). We use a *SecureRandom* for that, since it increases the entropy of the encryption. In line 17, we fill the IV byte array with random bytes.

To create a Cipher, we call *Cipher.getInstance(parameter)* in line 19, where *parameter* is a `String`, and must follow the pattern "*algorithm/mode/padding*". Here, *algorithm* specifies, which algorithm the Cipher should use. *mode* specifies, which block mode the algorithm should use. Obviously, this part of the parameter only applies for block ciphers with a mode of operation. To specify no mode, `None` can be used, as in, for example, "*RSA/None/OAEPWithSHA-512AndMGF1Padding*". The third part is the padding. As not all algorithms should be used with all padding strategies, this is also important to specify. For example, "*AES/ECB/PKCS7Padding*" is insecure, while "*AES/CBC/PKCS7Padding*" is secure. Another example is "*RSA/None/PKCS1Padding*" versus "*RSA/None/OAEPWithSHA-512AndMGF1Padding*", where the second is considered more secure than the first.

The implementation of each algorithm is provided by the *provider*. The providers vary depending on the specific Java environment. The provider can be specified by the second parameter for *Cipher.getInstance(...)*. However, this is generally not recommended, since the provider specified may not be available in another environment. If the second parameter is omitted, a default provider, which is defined by the environment, is used. Different providers may use different defaults, or even provide different sets of algorithms. For this reason, the contract of *Cipher.getInstance(...)* can vary between providers. In general, only the "algorithm/mode/padding" pattern is enforced. This leads to workarounds, where providers choose different approaches. For example, "*RSA/None/OAEPWithSHA-512AndMGF1Padding*" is perfectly legal when using the *Bouncy Castle* [BC] provider, but it throws an

exception when using the *SunJCE* [Orab] provider, as it expects "*ECB*" instead of "*None*". Still, both inputs would lead to the same configuration, as the mode is unused anyway. Another example, where changing the provider changes the semantics, is default values. A user can omit the mode and padding substring, and only pass the algorithm, as in *Cipher.getInstance("algorithm")*. For example, *Cipher.getInstance("AES")* is valid. However, this is considered a bad practice for multiple reasons. First of all, some providers will default this to "*AES/ECB/PKCS1Padding*", which is considered insecure. Also, if data is encrypted using one provider, and decrypted using another, the decryption may fail due to incompatible modes or paddings. If an algorithm, or padding is not supported by the provider, a `NoSuchAlgorithmException`, `NoSuchPaddingException` will be raised respectively.

On the Cipher instance, we then call *Cipher.init(...)* in line 20. This is used for setting up the more specific configuration, for example, the key size. Also, *Cipher.init(...)* has a parameter for the encryption mode, which decides if the Cipher instance encrypts or decrypts.

The cipher text is generated by *Cipher.doFinal(plaintext)* in line 22, where *plaintext* is the data in bytes to be encrypted. In the same way, the data can be decrypted, if the Cipher was initialized using the decryption mode. It is sometimes preceded by one or more *Cipher.update(...)* calls, if the data is encrypted in parts. This is useful if, for example, the data is provided via a stream of data and we do not want to intermediately save the unencrypted data.

Lastly, the application must in some way save the Secret Key and IV used for the encryption to be able to decrypt the data. The decryption is relatively simple. In lines 38 and 39, we create a *Cipher* instance with the same configuration used for encryption and initialize it using *Cipher.DECRYPT_MODE*, and the Secret Key and IV we saved from the encryption. To decrypt the data, we call *Cipher.doFinal(...)* with the encrypted data in line 40.

As can be seen in the example, several exceptions can occur in this process. They each represent either that a configuration is not available from the current provider, or if incompatible configurations are used.

In this thesis, we use a specification to identify incorrect usage of *Cipher*, which we introduce in the next section, Section 2.2.

2.2 Specification

In Section 2.1, we describe the usage patterns of *Cipher* from a more general point of view. However, as we require as much detailed information as possible on how *Cipher* should be used for our manual review in Chapter 4, this general usage description is not sufficient for consistently evaluating *Cipher* usages.

In this thesis, our main source for checking the correct usage of *Cipher* is a specification by Krüger [Krü], which is not yet published. Hence, we cannot show it in full detail here, but instead provide some examples to give an overview of its structure. The specification states formal rules for method sequences, secure configurations, and details, such as, if a configuration requires random parameters. An example rule from the specification,

$$\begin{aligned} &\text{fst(transformation) in \{AES, Blowfish, DESede, RC2\}} \\ &\quad \Rightarrow \text{trd(transformation) in \{NoPadding, PKCS5Padding, ISO10126Padding\}} \end{aligned} \tag{2.1}$$

states that, if *AES*, *Blowfish*, *DESede*, or *RC2* is the algorithm, the padding must be one of *NoPadding*, *PKCS5Padding*, *ISO10126Padding*.

As described in Section 2.1, correct usage may vary between providers. Hence, to provide a general specification, Krüger works on multiple rules, where some rules are specific to one provider. The rules

```java
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.IvParameterSpec;

public class AESEncryption {
  public byte[] encrypt(byte[] dataToEncrypt) {
    byte[] encryptedData = new byte[0];
    try {
      final int AES_KEYLENGTH = 256;
      KeyGenerator keyGen = KeyGenerator.getInstance("AES");
      keyGen.init(AES_KEYLENGTH);
      SecretKey secretKey = keyGen.generateKey();


      byte[] iv = new byte[AES_KEYLENGTH / 8];
      SecureRandom prng = new SecureRandom();
      prng.nextBytes(iv);

      Cipher cipher = Cipher.getInstance("AES/CBC/PKCS7Padding");
      cipher.init(Cipher.ENCRYPT_MODE, secretKey, new IvParameterSpec(iv));

      byte[] cipherText = cipher.doFinal(dataToEncrypt);
      /* ... save secretKey and iv values ... */
    }
    catch (NoSuchAlgorithmException noSuchAlgo) { /* ... */ }
    catch (NoSuchPaddingException noSuchPad) { /* ... */ }
    catch (InvalidKeyException invalidKey) { /* ... */ }
    catch (BadPaddingException badPadding) { /* ... */ }
    catch (IllegalBlockSizeException illegalBlockSize) { /* ... */ }
    catch (InvalidAlgorithmParameterException invalidParam) { /* ... */ }

    return ecryptedData;
  }

  public byte[] decrypt(byte[] encryptedData, SecretKey secretKey, byte[] iv){
    byte[] decryptedData = new byte[0];
    try {
      Cipher cipher = Cipher.getInstance("AES/CBC/PKCS7Padding");
      cipher.init(Cipher.DECRYPT_MODE, secretKey, new IvParameterSpec(iv));
      byte[] decryptedData = cipher.doFinal(encryptedData);
    }
    catch (NoSuchAlgorithmException noSuchAlgo) { /* ... */ }
    catch (NoSuchPaddingException noSuchPad) { /* ... */ }
    catch (InvalidKeyException invalidKey) { /* ... */ }
    catch (BadPaddingException badPadding) { /* ... */ }
    catch (IllegalBlockSizeException illegalBlockSize) { /* ... */ }
    catch (InvalidAlgorithmParameterException invalidParam) { /* ... */ }
  }
}
```

Figure 2.1: An example using *Cipher* to encrypt and decrypt data using *AES* in *CBC* mode with *PKCS7* padding.

we present as examples in this thesis are taken from the specification of the `SunJCE` [Orab] provider in *Java version 8*.

2.3 Possible Problems

In this section, we discuss possible reasons why *Cipher* may be misused as often. These observations are based on previous reports, as well as our own experience, and we do not suspect they are thorough. As such, this section is meant to give a better understanding of the inherent problems in *Cipher*'s design. In Section 4.4, we will take our review results into account, and discuss the issues we find in more detail.

2.3.1 Method Overloads

The *Cipher* API uses method overloads to take all functionality into account. For example, if using an algorithm without an *Initialization Vector (IV)*, the user could still use the *Init* override with IV. This may lead to unintentional use of default IVs. Since there are multiple dependencies between actual parameters, we suspect users may often accidentally use incompatible combinations.

2.3.2 Inconspicuousness

Often, misuses can be found by testing. Either they produce some error, or some unexpected behavior. For *Cipher*, especially when it comes to security vulnerabilities, this is not the case. Consider, for example, a test that encrypts and decrypts some data. If *Cipher* uses *DES*, the encryption will be insecure. To find this vulnerability, the test would have to implement an attack pattern. However, even if it did implement such an attack, which by itself is quite unrealistic, considering the number of attacks there are, and the time it would need to implement such an attack, it would need a very large amount of input data. The runtime of such a test suite would be unacceptable for any efficient development process.

2.3.3 Lack of Knowledge

In their survey, Nadi et al. [NKMB16] find that most developers struggle with cryptography APIs. Very commonly, developers are not confident in their knowledge of cryptographic concepts. *Cipher* amplifies this problem, since its architecture leaves no hints about security of usages. As explained in this chapter, *Cipher*'s architecture is based on the provider pattern. This cuts any direct access to the cryptography provider used. Hence, the user cannot access support by the implementing party, such as Javadoc [Oraa], or other documentation. This hides a lot of information from the developer, thus we suspect it is a major source of the insecure usage of *Cipher*.

Conclusion

In this chapter, we explain how *Cipher* can be used in Section 2.1. We have seen that *Cipher* implements the provider pattern. Thus, its behavior varies between different providers. However, we describe the over-arcing pattern of `Get→Init→Final`, and to configure *Cipher* to use a specific algorithm by passing the "*algorithm/mode/padding*" String to *Cipher.getInstance(...)*.

We then shortly introduce the specification by Krüger [Krü] we use in this thesis in Section 2.2.

Lastly, we discuss some possible problems of *Cipher* in Section 2.3. We find that the provider patterns brings a lot of method overloads with it. We suspect this could be a problem for developers, as usage patterns can vary a lot depending on the parameters used. Another problem we recognize is that insecurity is unlikely to be noticed by the user. Hence, we suspect to frequently find insecure *Cipher* configurations, and especially suspect these configurations to persist. This is escalated further by the lack of knowledge about cryptographic concepts past research [NKMB16] has shown. As developers often do not know, if a configuration is sufficiently secure, they may often use less secure, or even incompatible configurations, without noticing.

Thus, we find that there are several reasons, why *Cipher* is frequently used incorrectly. In the next chapter, Chapter 3, we introduce API-misuse detection as a promising tool to support developers using *Cipher*.

3 API-Misuse Detection

In this chapter, we explain what API-misuse detection is, how it works, and why we think it can help developers to correctly and securely use *Cipher*.

With an ever faster growing amount of APIs, developers often struggle to use these APIs as they intend. In comparison to documentation, and other resources, API-misuse detection seeks to support developers more directly. The approach is to invent tools, so called API-misuse detectors, which evaluate the user's codebase to warn about potential misuses. An API misuse is a violation of *usage constraints* of the API. In the case of *Cipher*, when a user omits the initialization, we call this a misuse, as it is a *usage constraint* of *Cipher*.

API-misuse detectors commonly use patterns to find misuses. They generate the patterns in a mining stage, and, using these patterns in the detection stage, they use differing approaches to compare the users code with their mined patterns. Pattern models vary between detectors, as each model is crafted to suit exactly the detection approach implemented by the detector. However, with sufficient information, all detectors can mine their respective models from the same sources. The only exception is, that detectors are implemented to target different programming languages. For example, we cannot easily compare the capabilities of `Colibri/ML` [Lin07] with the capabilities of `GROUMiner` [NNP+09], since `Colibri/ML` targets C, while `GROUMiner` targets Java code. Also, some detectors rely on source code for their pattern mining, but some use bytecode. Still, using MuBench [ANN+16], since it provides both, source code and bytecode, we can directly compare these detectors, whereas comparing detectors targeting different programming languages is harder to solve. In this thesis, we focus on API-misuse detectors which detect on Java, as *Cipher* is also a Java API. We describe the API-misuse detection approaches we evaluate in this thesis more thoroughly in Section 5.4.

Detectors can generally evaluate different aspects of usages, depending on their approach. The most prevalent examples are method sequencing problems, such as missing, superfluous or misplaced method calls, and control flow problems, such as missing and superfluous iterations or branching. For this, they rely on their mining stage to successfully spot the correct patterns. Often, the patterns are decided by frequency only, and less frequent derivations of the pattern are considered suspicious.

3.1 Classifying *Cipher* Misuses

As the comparability of *Cipher* and other APIs is yet unclear, we inspect, if *Cipher* misuses can be classified like other misuses. To see, if *Cipher* misuses are comparable to general misuses or fall into very specific categories, we need a taxonomy for misuses.

MuBench uses the *API-Misuse Classification (MuC)* [ANN+16], a taxonomy of API misuses. A MuC class is a composition of three parts:

The main part is the API element, for which a constraint was violated, for example `method call`, `exception handling`, or `condition`.

The prefix describes how the constraint is violated. There are three prefixes in MuC: `missing`, `misplaced`, and `superfluous`. `missing` means, the specified element should be in the usage, but is not. An *Cipher* example for this is, when a new *Cipher* instance is generated, but *Cipher.doFinal(...)* is called, without initializing the instance at all. In MuC terms, this kind of misuse is a `missing method`

call. In the same scenario, using the initialization twice is a superfluous method call. In case of a misplaced method call, the initialization is called, but *Cipher.doFinal(...)* is called beforehand.

The second prefix is not always used, as in, for example, missing method call. When used, it specifies a subcategory for the element. This is mainly used for condition, as it is a broad category, thus would not be comparable very well. For example, we differentiate between conditions on the value of an input, expressed as *missing/misplaced/superfluous* value condition, and conditions on the synchronization of code, which we classify as a *missing/misplaced/superfluous* synchronization condition. Without the second prefix, we would treat these misuses as the same, even though they have very different natures.

Due to its classification of misuses, MuC can also be used to compare API-misuse detector capabilities. Our goal in this section is to see, if MuC can accurately represent misuses of *Cipher*. First, to check the completeness of MuC, we compare it to a taxonomy of API directives by Monperrus et al. [METM12].

For each directive, we assign a MuC class representing violations of the directive. This cannot be done for all directives, since some do not have directly corresponding violations, for example, a Null Allowed Directive only allows a specific scenario and thus can not be violated. Also, some violations cannot be found by static source code evaluation, hence they are out of the scope of API-misuse detectors.

The first group of directives are called *Method Call Directives* and represent directives used in method documentation. We describe the directives give matching MuC classes one by one.

Not Null Directive stating that a specific method parameter must not be null. A violation of this directive can be classified as a *missing null check*, if the caller does not intentionally set the value for the parameter. For example, if he gets a value from a database, but the database may return null. However, if the caller directly sets the parameter value, or even passes null itself, *missing value/state condition* applies, since the directive sets the restriction on the parameter's condition not to be null.

Return Value Directive stating a required property on the returned object or value. For example, the returned *List* is never empty. When violated by the callee, it can be classified as a *missing value/state condition*. This directive can not be violated by the caller, since it has no control over the returned object or value.

Method Call Visibility Directive stating the visibility of a method. For example, "*Constructor only used in deserialization, do not use otherwise.*" This may also be an explicit permission to call a method, thus can not always be violated. Since these directives must declare a specific context in which they apply, a violation can only appear in the given context. Using the example from above, the context would be "*[...] in deserialization [...]*". Only when we are not in the context of deserialization, the directive can be violated. Hence, we can class the violation as a *missing context condition*.

Exception Raising Directive stating the exception being thrown, the situations in which an exception is thrown, or if a specific exception should never be thrown. Violations are classed as *exception handling*. They can be both, *missing exception handling*, but also *superfluous exception handling*. This depends on the directive. It may state an exception should never occur, meaning a violation can only be a *missing exception handling*. On the other side, it may state an exception should always occur in a specific context. In case of a violation, this is a *superfluous exception handling*, since the exception was handled, even though it should not have been.

Null Allowed Directive stating that a specific method parameter may be null and the semantics of that situation. As mentioned, this directive does not correlate to a violation, hence we skip it in this comparison.

String Format Directive stating the allowed format of a string that is passed in as a method parameter. A violation of this directive falls into the *missing value/state condition* class, as the directive states a restriction on the string's value.

Number Range Directive stating a certain range in which the value of a method parameter should be. This is only applicable to types representing numbers, for example, byte, int, or float. As for string format directives, *number range directives* restrict the allowed values of the parameter, thus a violation is a *missing value/state condition*.

Method Parameter Type Directive stating a restriction on the allowed type of a method parameter. For example, "*@param obj [typed as Object] the object to be serialized (must be serializable) [i.e. must implement the interface Serializable]*". Violations can be classified as *missing value/state condition*, where the condition is on the type of the method parameter, being part of its state.

Method Parameter Correlation Directive stating inter-dependencies between one or more parameters of a method. For example, "*If the given key is of type java.security.PrivateKey, it must be accompanied by a certificate chain certifying the corresponding public key.*". Using one of the correlated parameters as a triggering context for the other, we can classify a violation as a *missing context condition*. Using the given example, the directive states the context when *java.security.PrivateKey* is passed. If the usage does not pass the correlated parameter, in this example the public key certificate chain, correctly, this context is missing, thus violating the directive.

Post-Call Directive stating what immediately must be done with the returned object. When violated, *missing method call* is applicable. For example, a user should call *Cipher.init(...)* on a Cipher instance before calling any other method, such as *Cipher.doFinal(...)*, or *Cipher.update(...)* on it.

Another group of directives are *Subclassing Directives*. Since API-misuse detectors do not handle subclassing misuses, MuC does not cover violations of these directives.

The third group are *State Directives*. They set requirements on the internal state of receivers of a given method call. As before, we compare them one by one.

Method Call Sequence Directive specifying an object usage protocol. They also specify the order of method calls. These directives directly correlate to all *method call* classes. For example, the method call sequence for Cipher should be `Get→Init→Final+`, meaning we get a new Cipher instance with *Cipher.getInstance(...)*, initialize it using the correct *Cipher.init(...)*, which may also include *Cipher.update(...)* as needed, and use *Cipher.doFinal(...)* to generate a number of cipher texts. If a usage misses *Init*, the violation is a *missing method call*. On the other hand, if it calls multiple *Inits*, we would classify them as *superfluous method call* violations. Calling *Final* then *Init* is classified as a *misplaced method call*.

Non Call-based State Directive states a requirement on the application state that is not expressible as a method call sequence. For example, "*If this pathname denotes a directory, the directory must be empty in order to be deleted.*". These directives state a specific context, the application state, as a condition. Hence, a violation is classified as *missing context condition*. For the example directive, a violation would be using a pathname, where the denoted directory is not empty. This violation misses the context stated by the directive.

Then, there are two kinds of directives, which fall into none of these groups.

Alternative Directive stating that there are alternatives to a given API element. As in the Null Allowed Directive, this is not a restriction and does not have a corresponding violation.

Synchronization Directive stating some information regarding the impact of concurrency on an API element. For example, this may state if an operation is thread-safe or not, e.g. "*Note that FixedSizeMap is not synchronized and is not thread-safe. If you wish to use this map from multiple threads concurrently, you must use the appropriate synchronization.*". The correlating MuC violation types are *missing synchronization*, and *superfluous synchronization*. In the example above, a violation would be assuming *FixedSizeMap* is synchronized, rendering it a *missing synchronization*.

Finally, *Miscellaneous Directives* are all directives which do not fall into any specified classes. As these are very rare and specific, we exclude them from this comparison.

For *Cipher*, some of these directives can be used to describe its usage:

The *Not Null Directive* is applicable, as the none of the parameters used should be `null`. The API uses method overloads instead of `null` for default or unused parameters. For example, if we use a configuration which requires no Initialization Vector (IV), the user should use an overload of *Cipher.init(...)* without the IV parameter, instead of passing `null` as an actual parameter.

However, another directive can be used to document this behavior, namely the *Method Call Visibility Directive*. Using this directive, we can express the restriction from our example, if, for each overload, we state the context in which the overload should be used.

As exceptions are raised when incompatible or unavailable configurations are used, the *Exception Raising Directive* is also applicable. All of these exceptions should be handled by the user, hence they should be informed about them. Examples are `NoSuchAlgorithmException`, `InvalidKeyException`, `IllegalBlockSizeException`, and `InvalidAlgorithmParameterException`.

As for the parameter of *Cipher.getInstance(...)*, a *String Format Directive* can be used to describe the required "*algorithm/mode/padding*" format.

For the *Cipher.getInstance(...)*, *Cipher.init(...)* sequence, a *Post-Call Directive* exactly describes this constraint. On the *Cipher* instance returned by *Cipher.getInstance(...)* the user should always call an overload of *Cipher.init(...)*

Somewhat redundant with the Post-Call Directive, the `Get→Init→Final` sequence can be directed using a *Method Call Sequence Directive*.

According to Monperrus et al. [METM12], the *Not Null Directive* is used abundantly, the *Method Call Sequence Directive* is used commonly, the *Method Call Visibility Directive* and the *Exception Raising Directive* are used frequently, and the *String Format Directive* and the *Post-Call Directive* are used rarely.

Judging by the API directives applicable for *Cipher*, we expect it to be comparable to other APIs. The only rarely used directive is the *Post-Call Directive*, which can also be expressed by the commonly used *Method Call Sequence Directive*, which is probably more relevant for *Cipher*, as it can show the full usage of `Get→Init→Final`, instead of only `Get→Init`.

3.2 What API-Misuse Detection Can Do

In Chapter 2, we show a several usability problems of *Cipher*. In this section, we discuss how API-misuse detection can help developers using *Cipher*. We will focus on the capabilities of the four detectors `DMMC` [MM13], `GROUMiner` [NNP+09], `Jadet` [WZL07], and `Tikanga` [WZ11], as these are the detectors we evaluate in this thesis.

3.2.1 Lack of Knowledge

In Section 2.3.3, we see that developers often struggle with cryptographic APIs due to their lack of knowledge about cryptographic concepts.

If API-Misuse Detection can find issues in *Cipher* usage, it should also provide the user with some insight on how to fix the usages. This could potentially help developers to find secure configurations, even though it will not assist them in understanding the underlying cryptographic concepts. Hence, API-Misuse Detection cannot completely solve this issue, but it may bring it more into focus.

3.2.2 Method Overloads

As described in Section 2.3.1, *Cipher* relies on method overloads in multiple places. For example, there is *Cipher.init(...)* with an Initialization Vector (IV) and without IV.

API-Misuse Detection can generally warn the user about these issues and help to solve them. However, the detectors we evaluate in this thesis do not differentiate enough between method overloads.

3.2.3 Inconspicuousness

In Section 2.3.2, we discussed that most *Cipher* misuses concerning security cannot be found by testing.

API-Misuse Detection could help a lot in this area, as it can warn developers about these misuses, and make them more visible. Especially insecure configurations, which are solely based on the one parameter of *Cipher.getInstance(...)*, should be identifiable by such a tool. Unfortunately, since the detectors we evaluate in this thesis do not consider parameters, we suspect they cannot find these misuses. However, parameters could be taken into account by API-misuse detection, which would potentially enable detectors to warn about insecure configurations.

Conclusion

At the start of this chapter, we introduce API-misuse detection as a tool to find violations of API constraints.

We then compare a taxonomy of API directives to the *Misuse Classification* (`MuC`) to see, if we can relate API directive and possible *Cipher* violations in Section 3.1. We find that the most common *Cipher* violations are, in fact, violations of constraints which can be described by abundantly to frequently used directives. Thus, we deduct that comparable misuses could probably be found for other APIs and are not only specific to *Cipher*.

In Section 3.2, we refer to the possible problems we identify in Section 2.3, and discuss how API-misuse detection can help developers to notice these issues. We find that API-misuse detection is very promising, as the main problems of *Cipher* are the inconspicuousness of insecure usages and the lack of knowledge about cryptographic concepts. API-misuse detection could warn developers when they would not be able to notice these issues otherwise.

In the next chapter, Chapter 4, we build a novel dataset of *Cipher* misuses, which we use in Chapter 5 to evaluate and compare the capabilities of four state-of-the-art API-misuse detectors.

4 Building a Dataset for MuBench

Previous work [EBFK13, NKMB16] reports that the *javax.crypto.Cipher* application programming interface (API) is misused regularly. However, this research is based on automated evaluation and survey studies. Since the main goal of this thesis is to evaluate the performance of misuse detectors on the Java Cryptography Extensions (JCE) using the benchmarking tool MuBench [ANN+16], we need data which fulfills the following minimal requirements:

- It must contain any number (greater than zero) of *Java source code* files.
- The Java source code files must be *compilable*.
- The file and method in which the misuse is *located* must be known.

As none of the available data is sufficient under these conditions, we first attempt to build a new dataset using some of the existing data as a source. We base our data on the field study results by Ziegler [Zie15], where he examined the security of disassembled android projects using automated tools. To emulate his findings we implement synthetic example programs.

4.1 Constructing a Synthetic Example

As the first approach to find *Cipher* misuses, we take a project from Ziegler's field study [Zie15]. We use the description of misuses he finds in Siemens-I-RAS, and write the minimal source code to represent these misuses. In the context of MuBench, these examples are considered *synthetic*. The source code of this example can be seen in Figure 4.1. The snippet contains two misuses:

In line 7, we see *SecretKeySpec* being initialized using a constant key "*RAS*". An attacker could easily find this key even from compiled source code, and this is especially critical if the code is published. Users should never put keys in a publicly available place.

The second misuse is in line 9 and 17. Here, *Cipher.getInstance("AES")* initializes the *Cipher* instance using the default configuration. This should not be done, since the provider decides the configuration. For one, the provider might use an insecure default configuration. Also, this might break decryption, if the decrypting party uses a provider with different defaults. Users should always provide a secure mode and padding to avoid these issues.

However, this synthetic example does not prove satisfactory. For one, it misses the usual context in which *Cipher* is used. We discuss this context more in Section 4.4. More importantly, continuing this approach, we would end up with four misuses—far from enough to conduct any conclusive experiments.

For these reasons, we need a better source for *Cipher* API misuses. We use Boa, which allows us to find *Cipher* usages on GitHub.

4.2 Mining *Cipher* Usages from GitHub

In Section 4.1 we show why the previously available data does not suffice to create a dataset for our experiments. Since we need more usages and would like to add whole projects to the dataset, we need

```
1  import javax.crypto.spec.SecretKeySpec;
2  import javax.crypto.Cipher;
3
4  public class Encrypting {
5    public static byte[] encryptWithKey(byte[] content) throws Exception {
6      // Using a constant key is insecure.
7      SecretKeySpec keySpec = new SecretKeySpec("RAS".getBytes("UTF-8"), "AES");
8      // Using a default configuration is insecur e. The user should always declare a secure
           mode and padding, i.e. "AES/CBC/PKCS5Padding".
9      Cipher c = Cipher.getInstance("AES");
10     c.init(Cipher.ENCRYPT_MODE, keySpec);
11     return c.doFinal(content);
12   }
13 }
```

Figure 4.1: Synthetic example representing *Cipher* API misuses described by Ziegler [Zie15].

a new source. Boa is a tool we can use for this purpose. It allows us to mine a GitHub Snapshot for projects with *Cipher* usages, which we can manually review.

Boa provides an infrastructure to mine open source projects. We can build visitors to traverse its data like a tree. Boa contains multiple snapshots from GitHub and Sourceforge. We use the full snapshot of GitHub September, 2015. It contains 7,830,023 projects, and 146,398,339 unique files.

First, we define a visitor using Boa's domain specific programming language, fully shown in Figure 4.2. The following will explain this visitor in more detail.

The visitor pattern Boa provides allows us to declare logic only for the elements we care about. Since we want to mine *Cipher* usages, the first step is to find *import* statements. Specifically, we seek files containing *import javax.crypto.Cipher*, or *import javax.crypto.**. The second, we use to maximize output size and variety, but it is not strictly required. We find the import statements in *ASTRoot*, which exists once for each *ChangedFile*. The filtering for these imports can be found in lines 15 to 23. There, we first compare the imports by simple string comparison. On match, we add the files GitHub URL to our output set. To build the URL, we need the project URL, which is given by the root input in line 1. Additionally, we need the revision ID, which we obtain in line 9, in the *Revision* node. As this runs before we enter each revision, the *revision* variable is already updated, when entering *ASTRoot*. The last part of the URL is *file.name*, which contains the full file path. The current file is set in line 12, similar to the current revision.

Running this code, we observe two problems:

First, we get a lot of redundancy in the output, as the same *ChangedFile* can be in multiple revisions, or even in multiple forks of a project. To counter this, we add a *files* set in line 4. This set contains all files of a project we already processed, as they added in line 14, after we processed it. Then, before we enter a new file, we check if the file is already known.

The second problem is that a lot of our output is test code. As this code serves a completely different purpose, we would like to exclude it as best as possible from our evaluations. Since files containing test code should by convention have "*Test*" or "*test*" in their name, we implement an empirical test code filter by case insensitively checking, if a file name contains "*test*".

For both cases, we check in line 11 if they occurred. `stop` means, we stop traversing further down, and instead skip this subtree.

This Boa program generates an output of 1,573 URLs.

```
p: Project = input;
out: output set[string] of string;

files: set of string;
revision: Revision;
file: ChangedFile;

visit(p, visitor {
  before r: Revision -> revision = r;
  before f: ChangedFile -> {
    if (contains(files, f.name) || match("test", lowercase(f.name))) stop;
    file = f;
  }
  after f: ChangedFile -> add(files, f.name);
  before astRoot: ASTRoot -> {
    imports: = astRoot.imports;
    foreach (i: int; def(imports[i])) {
      if (imports[i] == "javax.crypto.Cipher" || imports[i] == "javax.crypto.*") {
        out[p.name] << p.project_url + "/blob/" + revision.id + "/" + file.name;
        stop;
      }
    }
  }
});
```

Figure 4.2: Boa program to mine *Cipher* usages.

Using this output, we manually review a random sample to find misuses for the dataset, as described in Section 4.3.

4.3 Reviewing *Cipher* Usages

Using the results from mining GitHub in Section 4.2, we can now perform a manual review. For this review, we choose random projects until a threshold of 30 misuses is reached. As described in the introduction of this chapter, MuBench also requires us to supply an automated build process for each project. We consider this while counting projects for the new dataset.

We use a specification by Krüger [Krü] for our manual review, which is described in Section 2.2. The specification contains formal rules for *Cipher*. An example for these rules is

$$\begin{aligned} &\text{fst(transformation) in \{AES, Blowfish, DESede, RC2\}} \\ &\quad \Rightarrow \text{trd(transformation) in \{NoPadding, PKCS5Padding, ISO10126Padding\}}\,. \end{aligned} \tag{4.1}$$

This rule specifies that, if *Cipher* is initiated using *AES*, *Blowfish*, *DESede*, or *RC2* as its algorithm, then the padding mode must be one of *NoPadding*, *PKCS5Padding*, and *ISO10126Padding*. For example, using *Cipher.getInstance("AES/CBC/PKCS5Padding")* is correct under this rule, whereas *Cipher.getInstance("AES/CBC/OAEPWithMD5AndMGF1Padding")* is rated as a misuse. Also, *Cipher.getInstance("AES")* would be rated as a misuse, since with no information about the mode (snd) and padding (trd), we must assume that either insecure or incompatible defaults are possible, as described in Chapter 2.

We reviewed 26 projects to find a total of 30 *Cipher*-specific misuses. 18 projects violate one or more rules. To get an overview of the dataset, we look at the most frequently violated rules. We find nine violations of the rule

$$\text{fst(transformation) in } \{\text{AES, Blowfish, DESede, } \dots\}\,, \tag{4.2}$$

meaning that, in these cases, *Cipher* uses an insecure algorithm for encryption.

In 13 cases, algorithms are either used with default, or with insecure settings for block mode and padding. These violations are split into multiple rules, as they occur in the context of different algorithms, i.e. AES, RSA, and Blowfish.

Next, we look at less frequently violated rules. Notably, the order of method calls is rarely a problem. The order of method calls on a *Cipher* instance is specified as

$$\begin{aligned}&\text{ENFORCE ORDER}\\&\quad \text{Get, Init, Final+ .}\end{aligned} \tag{4.3}$$

Where *Gets* represents overloads of *Cipher.getInstance(...)*, and Init represents overloads of *Cipher.init(...)*. This is to enforce that instances are always obtained by the provider method. Subsequently, to initialize the Cipher instance, the user then needs to call the correct initialization method, followed by update methods depending on the algorithm used. Lastly, the user can call *Cipher.doFinal(...)*, here represented by *Final+*, to generate the cipher text. As *Init* does not ensure a full initialization of the state, calling *Init* multiple times on the same *Cipher* instance must be considered a misuse.

We find multiple *Init* calls in only two usages, suggesting this is not a critical part of the *Cipher* API. We find that most often, *Cipher* is initialized using insecure values, and the control flow misuses, such as superfluous method calls or missing initialization are much less frequent. In Section 4.4, we investigate thoroughly why *Cipher* is misused as frequently.

Using the detailed information we gained about each misuse, we have already covered most prerequisites for MuBench, described at the start of this chapter. We can checkout the complete source code of each project. We know the exact location in terms of class and method of the misuse. We identify the exact rules violated, giving us a formal description of the misuse. Yet, we still need compiled sources.

Since we assume that newer versions often use a better automated build process, we look for the newest project versions which still contain the misuses from the dataset. In these project versions, we try to identify if build tools are used by the projects, or, if not, how we could setup an automatic build. We focus on `Maven` [Foub], `Gradle` [Gra], and `Ant` [Foua] since these tools are widely used and we can run the build process fully automated using them. Unfortunately, we find only four projects compile without additional work. For projects which do not use any build tools, we setup a basic `Gradle` build. At this point, we need more specific approaches. Hence, we check the error messages for each failing build. Some projects only have missing dependencies. We add these dependencies to their specific build configurations, and add commands to the build process to remove some hard coded paths. We get a dataset of 10 misuses over 8 projects.

4.4 Why *Cipher* is Frequently Misused

In Section 4.3, we find *Cipher* being misused in 18 of 26 reviewed projects, which is a rate of 69.2%. In this section, we discuss possible reasons why the misuse rate of *Cipher* is as high as we observe.

```
before expr: Expression -> {
  if (expr.kind == ExpressionKind.METHODCALL) {
    is_cipher_getinstance := expr.expressions[0].variable == "Cipher" &&
                             expr.method == "getInstance";
    if (is_cipher_getinstance) {
      if (len(expr.method_args) < 1) stop;
      arg := expr.method_args[0];
      out << uppercase(get_arg_value(arg))
    }
  }
}
```

Figure 4.3: Boa-program extension to mine *Cipher.getInstance(...)* parameters.

4.4.1 Misleading Documentation

Most *Cipher* documentation uses insecure configurations for *Cipher* when usage examples are given. This is very misleading for developers, as they tend to trust the documentation and, in this case, they are implicitly expected to change the example to their needs. Even in the Android Developers documentation of *Cipher* [Goo], the example *Cipher.getInstance("DES/CBC/PKCS5Padding")* is a misuse due to *DES* being considered insecure. Since the documentation only uses this as an example and does not claim for it to be a secure usage, developers might not actually use the same configuration in their projects. However, we suspect there is a significant amount of *DES* uses, which might be caused by the documentation example. In our reviewed sample of 26 projects, we find 21 of 30 misuses being generally insecure configurations. To see, if the usage of insecure configurations, such as *DES*, is significant, we check what *Cipher* configurations are actually in use. Since we already created a Boa program to find classes using *Cipher* in Section 4.2, we can extend this program to find all *Cipher.getInstance(...)* expressions. From these expressions, we output the used parameter. This extension is shown in Figure 4.3. For each expression, we check if it is a *Cipher.getInstance(...)* method call, and if so, we save the actual parameter's value in the output. Just taking the value as-is, we find the actual parameter value is often passed using fields or variables, giving us a lot of unknown values. Hence, to get as many concrete values as possible, we implemented *get_arg_value(...)* to recursively resolve variable and field references. Since we can only find static values, there are still cases where we cannot identify the actual value. For these cases, a dynamic evaluation would be needed, which is not feasible for such a large amount of project versions.

We then count the frequency of parameter values on the full dataset of GitHub in September 2015, which was also used in Section 4.2. In Figure 4.4, our results are shown. Of approximately 8,600 parameters, 1,710 were using *DES*. That is close to 20% of all usages, and makes it the second most used algorithm after the *Advanced Encryption Standard (AES)*, which occurs in about 40% of usages. We suspect its use in the documentation is a significant factor in this. Even though API documentation is not the main focus of this thesis, it seems critical in the case of *Cipher*.

4.4.2 Using Defaults

In general, we consider the use of default mode and padding as a misuse, as the resulting configuration heavily depend on the provider. For one, many algorithms are by default used in insecure configurations.

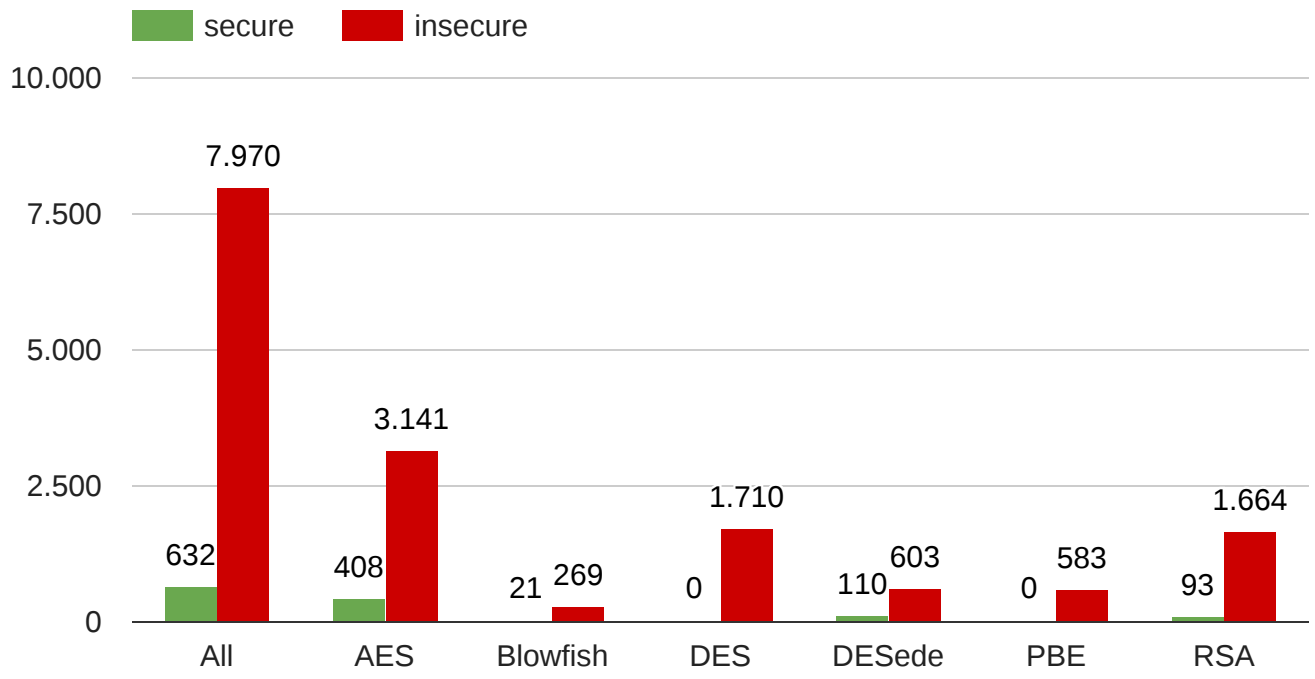

Figure 4.4: *Cipher.getInstance(...)* secure and insecure parameters for the 6 most used algorithms.

Additionally, a change of providers can lead to incompatibilities and even data loss. As an example, of a total of 3,549 *AES* usages, we find that 1,485 use the default configuration. With *Bouncy Castle* as the provider, *Cipher.getInstance("AES")* uses the *Electronic Codebook (ECB)* mode, which is considered insecure. We cannot estimate how often these problems surface in reality, but using a default configuration is, at the very least, a bad practice, and should be avoided in all cases.

4.4.3 Insecure Still Runs

In Figure 4.5, we see *Cipher* being configured insecure in 92.7% of usages. As discussed before, this is likely due to a combination of misleading documentation, and using defaults. Still, these arguments do not explain why most of the misuses persist through multiple project versions. We suspect these misuses occur due to a lack of cryptographic knowledge, and, hence, cannot be found by testing—often remaining unnoticed. The reason for this is using insecure algorithms and configurations, for example *DES*, produces neither errors nor warnings.

Consider a test that encrypts and decrypts "*Hello World!*". This would work fine, even with an insecure algorithm such as *DES*. Often, cryptographic attacks require a large amount of encrypted data to guess the key. If a test would implement such an attack to check for security, the developer could rarely even provide the required amount of data. Developers need tools to warn them when they use such a configuration. Additionally, we see insecure configurations occur frequently in all often-used algorithms, meaning this is not algorithm-specific, but rather a general problem.

4.4.4 Correct usage $\neq$ secure usage

As explained in Chapter 2, Cipher is designed to support all encryption and decryption procedures. Due to this breadth of available configurations, there are many insecure configurations provided. In our re-

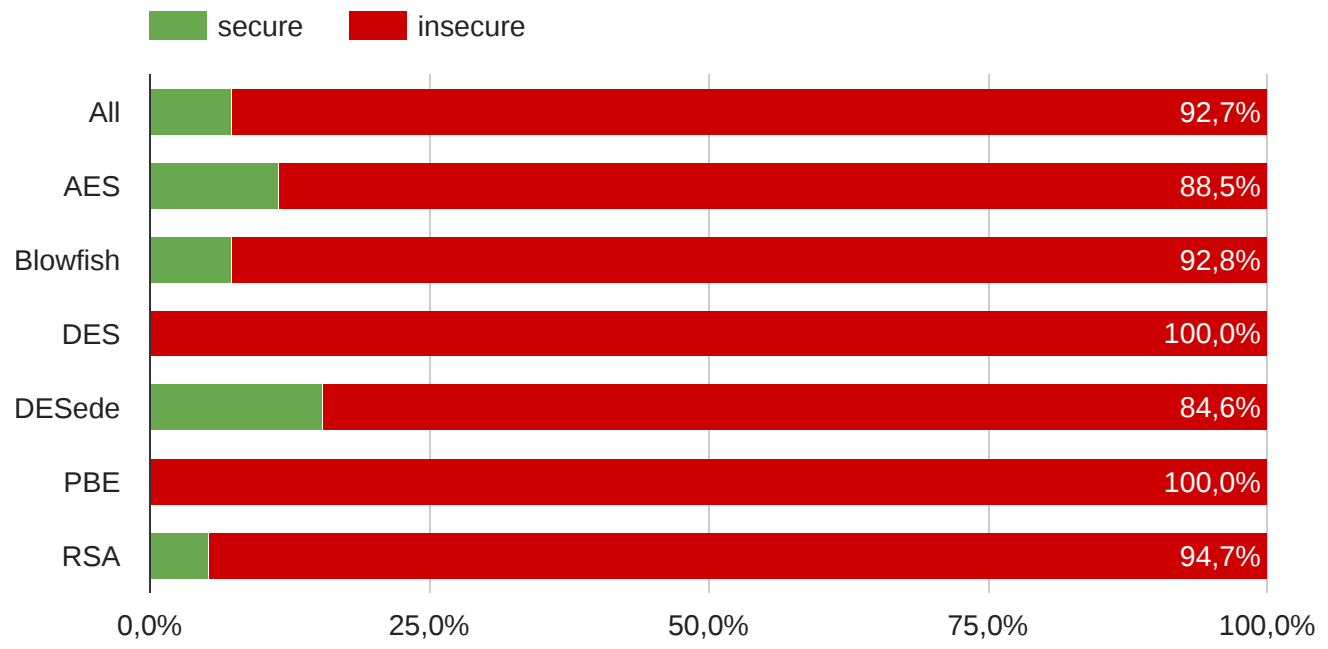

Figure 4.5: *Cipher.getInstance(...)* secure and insecure parameters for the 6 most used algorithms as percentages.

search, we observe a discrimination between correct usage concerning the API's constraints, and correct usage concerning the user's intention. In terms of Cipher, this is mainly the difference between correct usages, and secure usages. For example, *Cipher.getInstance("DES/CBC/PKCS5Padding")*, taken from the Android Developers documentation of *Cipher* [Goo], is technically a correct usage. The user requests a Cipher instance that uses the *Data Encryption Standard (DES)* in *CBC mode* with *PKCS5Padding*, and the provider will correctly return this instance. However, this does not take into account, that the user most likely wants the configuration to be secure. Hence, he will not receive any errors or warnings from the API, even though, in the user's sense, this is a misuse.

We suspect this kind of gap between the API's interest, and the user's interest, usually gets worse, the more generally applicable the API tries to get. In the context of Cipher, the concept to apply for all encryption algorithms, even if they differ a lot in inputs, outputs, and attributes, such as security, and runtime, has a major downside to its usability. In plain terms, the more paths the user can take, the more likely he is to get lost. Especially, since we know from past surveys, that developers are not confident in their cryptography knowledge [NKMB16].

4.4.5 API Misconceptions

As we see in Section 3.1, the most frequent problem we find in Section 4.3, where we manually review projects on GitHub, is the use of insecure *Cipher* configurations, i.e. insecure algorithms, such as *DES*, and default or bad block modes and paddings. However, as described in Section 2.3, the API generally allows these usages, as they do not produce exceptions and are consistent in their semantics. For this reason, there are no directives permitting them. We experience a gap between what the API considers incorrect and what the user would generally consider a misuse. Hence, we find a technically correct documentation of the API, which still contains security vulnerabilities, as discussed in Section 4.4. This

further marks the importance of tools to support developers on APIs like *Cipher*, as they cannot fully rely on documentation.

From the definition of API misuses, they are only based on API constraints. We have seen that, in the case of *Cipher*, these constraints are, in fact, not intended by the API. Hence, if we want misuses to also reflect the user's misunderstandings, we cannot define them only as violations of the API's constraints, but must consider violations of the user's constraints as well. This may not be feasible for all users, as some have very specific intents. For example, if a researcher would want to compare *DES* and *AES*, well knowing that *DES* is considered less secure, his usage would be fully intentional, thus no misunderstanding, and no misuse. However, we suspect that the majority of users use *Cipher* for securely encrypting data, thereby, using *DES*, misunderstand the API, and misuse it, considering their intention. In general, we must consider the user's intent, or at least the intent of the majority of users, when they use an API like *Cipher*, as, the more general-purpose the API, the less constraints it can enforce for its users.

To support these misuses in our considerations, we name them *API misconceptions*, where an *API misconception* is a *violation of a user's constraint*. We define the user's constraints as all criteria the user intends for his codebase to hold. Examples for user constraints are security, robustness, or performance. We notice, that, in general, these constraints are also criteria in quality assurance. A perfect validation of these constraints is probably not feasible for a fully automated tool, as they, obviously, vary a lot between individual users. Still, in the case of *Cipher*, we find an example, where holding the security constraint could very well be supported by detecting such API misconceptions. For example, warning the user that *DES* is considered insecure, and instead suggesting a secure configuration for a standard like *AES* could have a significant impact, as we see from the results in Section 4.4.3. In Figure 4.6, we see the frequency of API misconceptions for the 6 most used algorithms. This shows that for most configurations, the usages do not violate any API constraint, but are, instead, API misunderstandings. We should note, though, that this is just an estimation. We already discussed, that we cannot simply assume the security constraint for each user. As such, this estimation is very pessimistic, as many of these projects may either not be in use anymore, and some may deliberately use insecure configurations—reasons for which may be the use of very specific third-party providers, research, and others. However, even if only half of these potential API misconceptions are in fact misuses, that would still be a significant percentage.

4.4.6 Other Factors

Due to manual reviews of static code, we cannot consistently check rules such as correct key size, and secure randomness. These may also be sources of misuses, as past research found some misuses analyzing Android Java Bytecode [Zie15]. Further investigating this was not feasible in the scope of this thesis, but may prove that tooling support is required in these areas as well.

Conclusion

In this chapter, we creat a new dataset of *Cipher* misuses. Since we show in Section 4.1 that the naive approach of creating synthetic examples from descriptions of misuses from previous work does not yield the details we require for our purpose, namely missing the source code context of real-life projects, we looked for other approaches to obtain the data we need.

In Section 4.2, we used Boa [DNRN13] to mine GitHub for *Cipher* usages, which gives us enough project source code to build a new dataset from manual reviews. Using the specification we described in Section 2.2, we review 26 projects on GitHub to find 30 *Cipher*-specific misuses.

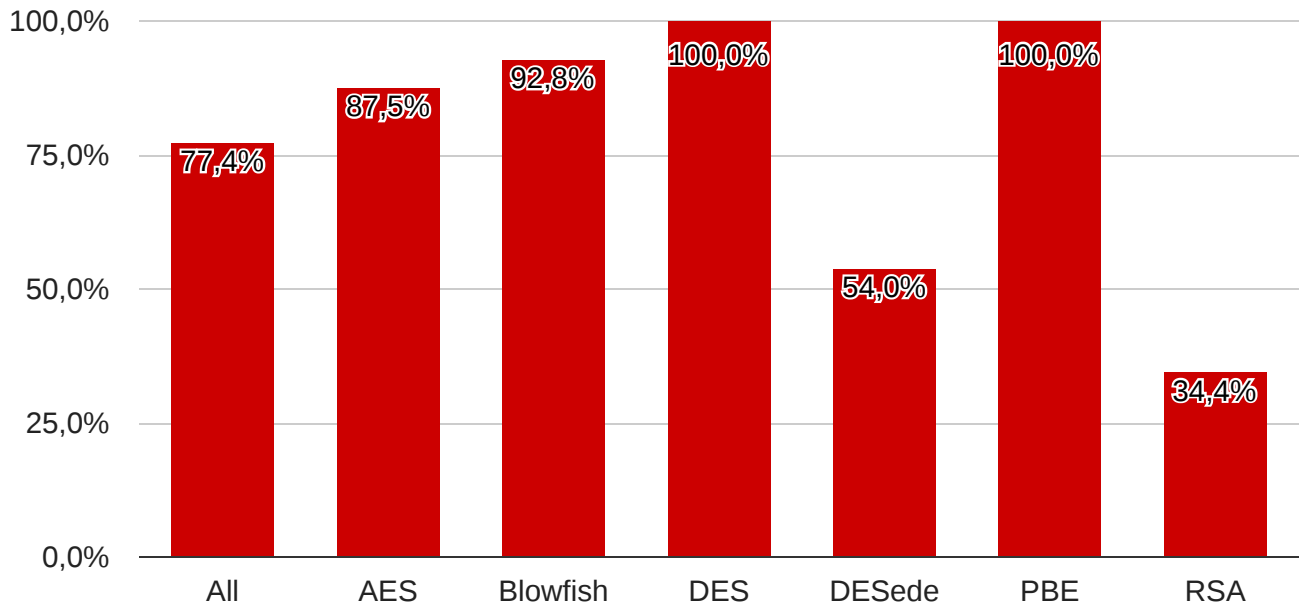

Figure 4.6: Percentage of *API misconceptions* considering security for the 6 most used algorithms.

We analyze the results of our review in Section 4.4 and discuss possible reasons for the misuses we found in Section 4.3. In the course of this discussion, we discover a problem concerning the discrepancy between the intention of the API and the intention of its users. For *Cipher*, we find that documentation does not consider security, but focuses instead on the high flexibility of the API, thus giving the user only minimal constraints. On the other hand, we experience a lot of security vulnerabilities in *Cipher* usages, which should be the main concern of most *Cipher* users. We also find that users rely on default configurations a lot, which we consider a bad practice, since it potentially introduces incompatibilities and security vulnerabilities, depending on the provider used in the Java environment the code is executed in.

Since we experience these misunderstandings so frequently, we introduce a new term to describe them, *API misconceptions*. We define an API misconception to be a violation of a user's constraint, which is derived from the previous definition of API misuses, being violations of API constraints. We observe that for *Cipher* the user constraint security is not shared well by the API. We conclude that this is the main source of *Cipher* misuses. Hence, we feel that the need for tooling support is very high in this area, as developers need a source of validation that takes their constraints into account.

In the next chapter, Chapter 5, we use MuBench, an API-misuse dataset and API-misuse detector benchmarking tool, to compare the capabilities of four state-of-the-art API-misuse detectors. We then evaluate these capabilities and discuss what future work can do to improve API-misuse detection on APIs such as *Cipher*.

5 Evaluating API-Misuse Detectors

As described in Chapter 3, tools which warn the programmer when they rate a usage as incorrect are one approach to help reduce API misuses in general. We call these *API-misuse detectors*. We focus on API-misuse detectors, as they are built for the specific purpose of finding API misuses, and, as such, we would like to inspect their performance on the often misused *Cipher*. Also, they were not yet evaluated on this API. Hence, our goal is to examine how current misuse detectors perform on the *Cipher* API and defer some suggestions on future improvements. To this end, we use the dataset created in Chapter 4 to extend MuBench [ANN+16], an API-misuse dataset and benchmarking tool for API-misuse detectors, and to evaluate API-misuse detectors on *Cipher*.

5.1 About MuBench

MuBench [ANN+16] is a benchmark for API-misuse detectors. It currently supports four API-misuse detectors, which we introduce in Section 5.4 and contains a dataset of about 100 API misuses.

Using the API-misuse dataset, MuBench runs the misuse detection on each project version, and collects the results automatically. It can also filter the results before the manual review by comparing the method in which the misuse is located to the method in which the detector suspects a misuse. This filtering cannot catch all false positives, since the detector may suspect a misuse at the correct location, but for the wrong reasons. For this reason, a manual review is needed to find the true-positives.

The collected results are uploaded to a review site, where they can be reviewed manually. The review site provides all available information for the manual review. Amongst other things, it shows the method in which the detector suspected a misuse, a pattern showing the correct usage if available, the misuse description, and why the detector suspects a misuse. The latter is part of the detector output and very specific to its detection approach.

The reviewer can enter his name and final evaluation on the site. We can then generate statistics from the results, i.e. found number of misuses, characteristics (missing value, missing method call, ...), and more.

5.2 Adding the Data to MuBench

In Chapter 4 we built a dataset containing misuses of *Cipher*. This data is based on GitHub projects and contains detailed information about each misuse. We would like to use this data to run the benchmark. MuBench requires the data in a specific form. It sorts the misuses first by project and second by version. An overview of the folder structure of MuBench can be seen in Figure 5.1.

For each project, we create a directory in the MuBench data folder, which will contain all information about the project and its misuses.

Inside this directory, we create a *project.yml* file. This file contains the project name, repository URL and type (MuBench currently supports GitHub, SVN and synthetic), and other general project information, e.g. the project's website.

Besides the project file, we create two directories. *versions* and *misuses*:

versions contains a subdirectory for each version of the project. MuBench requires this structure, since a project might have its misuses spread out across multiple versions.

Each folder in *versions* must have a *version.yml* file. *version.yml* contains the ID of its latest commit.

MuBench checks out the project versions from GitHub, and some misuse detectors require compiled code. For this reason, we need to give MuBench the commands for compilation, as well as some additional files if needed. This information is entered into the *version.yml* as well. Sadly, we can only use project versions which we can build in an automated fashion. Trying to compile these projects is very time consuming, so we targeted to add 10 projects. Counting only the 10 projects we can compile, we now have 14 out of the 25 misuses we found left.

misuses contains a subdirectory for each misuse of the project. Here, we put the misuse specific data. We create a *misuse.yml* file, containing all specific information about the misuse, e.g. a short description, what should be done to fix it, the location as in method and surrounding class, and the used APIs. This information is needed for manual reviews of the detector results, but also used by the benchmark to filter detector results for potential hits as they often generate a lot of output. This filtering process is not used in all experiments, but it is vital for true positive evaluations, as will be described more thoroughly in Section 5.5.

In addition to the misuse file, we create a *patterns* folder. This folder contains *.java* files with minimal compilable examples of correct usages. MuBench will pass this information to the detectors in some experiments to check if they find the misuse when they know the pattern that is violated. The experiments are shown more detailed in Section 5.5.

5.3 Extending MuBench to Run on Data Subsets

MuBench currently uses a dataset containing 100 misuses. These misuses are spread between many APIs. As we want to focus our experiment on *Cipher*, we need to implement a new feature for MuBench. We want to run it on a subset of misuses. Using this feature, we can specify a subset containing only the *Cipher* misuses we added in Section 5.2.

We add a new file in the data root folder, named *datasets.yml*. This file contains a list of datasets, which are all subsets of the complete data. A dataset is just a set of misuses. The misuses are uniquely named by *project-name.number*.

Using these new datasets, we add a new option to the command-line interface, called dataset. This option allows the user to run MuBench only on misuses in the given dataset. We create a new *Cipher* dataset and add the misuses from Section 5.2.

With this new feature in place, we can now run the experiment and review our results in the next section, Section 5.5.

5.4 Detectors

In this section, we shortly introduce the four API-misuse detectors currently supported by MuBench. They are all API-misuse detectors for Java, thus we can directly compare them. There are, in fact, more API-misuse detectors, but of the 12 detectors yet introduced in research, only 7 run on Java, and, of these, `CAR-Miner` [TX09b] and `Alattin` [TX09a] rely on Google Code Search [DiB15], which is not available anymore, and `DroidAssist` [NPVN15] only runs on Android Java Bytecode, which MuBench does not provide.

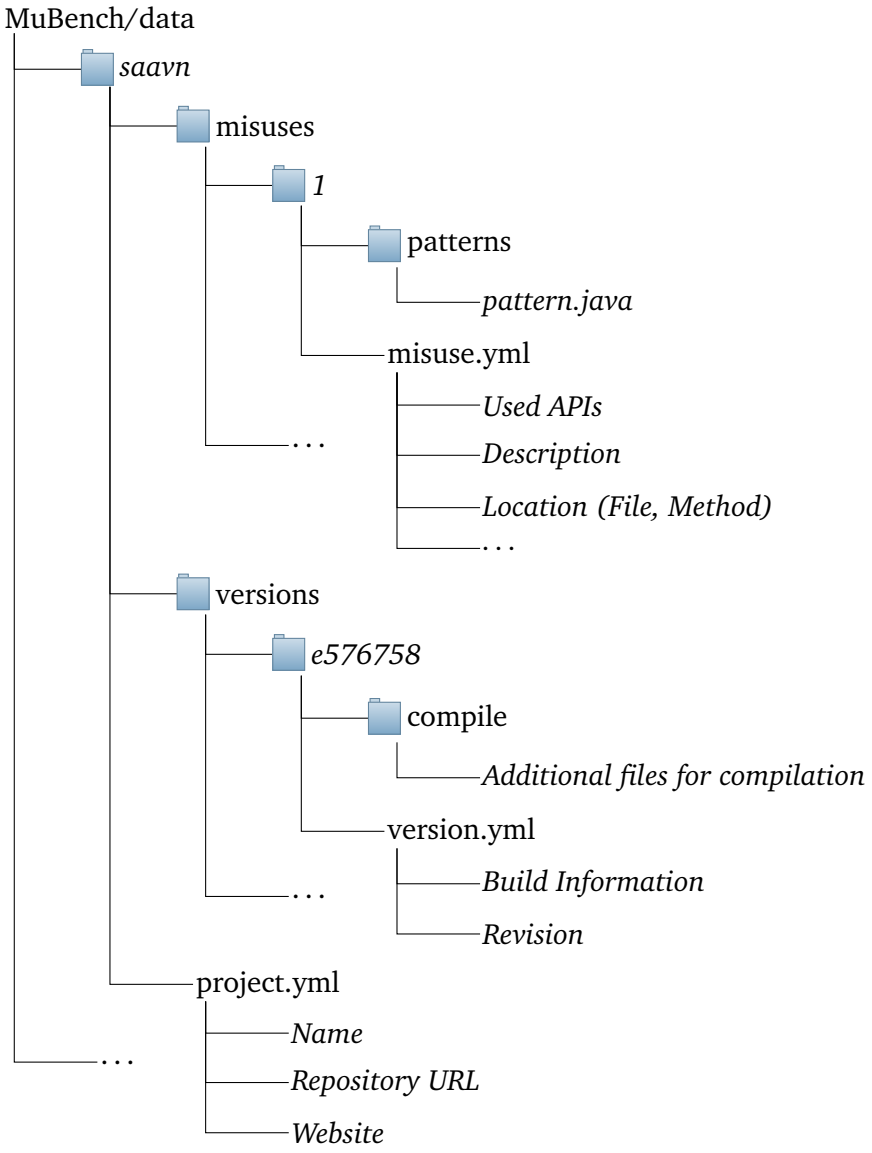

Figure 5.1: Example of the MuBench data structure

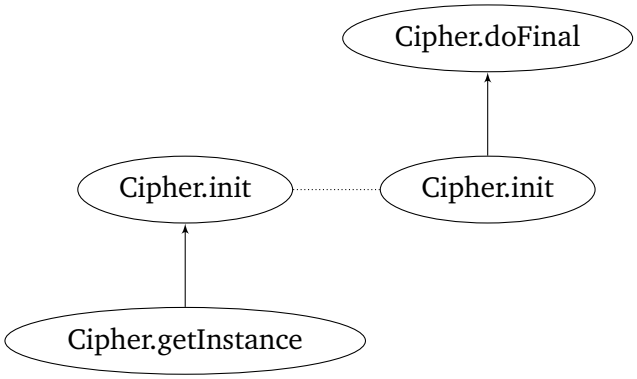

Figure 5.2: Two GROUMs are combinable, because they share a node.

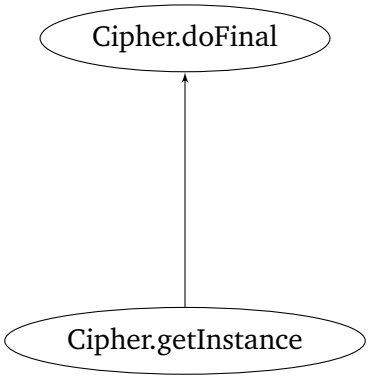

Figure 5.3: A violation, because it is missing the shared node from Figure 5.2.

5.4.1 DMMC

DMMC [MM13] is designed to find missing method calls. It uses sets of methods called on an instance of a given type to find type usages. It then compares these usages using the criteria of exactly similar, if the sets match, and almost similar, if one set contains exactly one additional method. If a set has many almost-similar, but only few exactly-similar sets, it is considered a violation.

As DMMC can, by design, detect missing method calls only, it can obviously find missing *Cipher.init(...)*. However, it cannot find insecure *Cipher* configurations.

5.4.2 GROUMiner

GROUMiner [NNP+09] creates a graph-based object-usage representation (GROUM) for each analyzed method. A GROUM is a directed acyclic graph, which contains nodes for method calls, branchings, and loops and edges for control and data flows. To find recurring patterns, GROUMiner looks for frequent subgraphs of sets of the mined GROUMs. When it finds that a majority of occurrences of patterns can be extended to a combined, larger pattern, it rates the occurrences that cannot be extended as violations.

For example, consider this common *Cipher* usage: the *Cipher* instance is obtained using *Cipher.getInstance(...)*, initialized using *Cipher.init(...)*, and, finally, the ciphertext is produced using *Cipher.doFinal(...)*. If GROUMiner finds the two patterns shown in Figure 5.2, it will combine them to a larger pattern, since they share the *Cipher.init* node. For this reason, GROUMiner can then detect the violation shown in Figure 5.3, since it misses the *Cipher.init* node.

Thus, violations detected by GROUMiner miss one node in their GROUM representation. In the case of *Cipher*, this means it could be able to detect missing method calls like *Cipher.init(...)*, or *Cipher.update(...)*, given it has the correct pattern. However, for method overloads, GROUMiner will not be able to distinguish between these contexts, hence, could not detect these misuses correctly, even if it were able to differentiate between the overloads.

Since GROUMiner does not save the actual parameters of usages, it cannot detect insecure configurations used in *Cipher.getInstance(...)*.

5.4.3 Jadet

Jadet [WZL07] is based on Colibri/ML [Lin07], which is a misuse detector for C. Colibri/ML mainly focuses on detecting missing method-calls.

Jadet builds a graph nodes for method calls on an object and edges for control flows. Using the graph, it deducts pairs of call-order relationships. Then, it uses these pairs for mining. If a usage misses 2 or less properties of a pattern and occurs at least ten times less frequently than the pattern, it is considered a misuse of that pattern. Due to the encoding of call-order relations, Jadet can detect misplaced calls as well as missing calls. Also, it may detect missing loops as a missing call-order relation from a method call in the loop header to itself.

Again, using the common sequence of *Cipher*, Get→Init→Final, Jadet can, for example, model the "*Init precedes Final*" relationship, denoted as *Init* $\prec$ *Final*. Hence, it can find violations where *Init* is missing or misplaced. However, as it does not consider parameters, it cannot detect insecure configurations of *Cipher*. It also suffers from the ambiguity of *Cipher*. As Jadet builds its relationships per type, it cannot differentiate between configurations. Hence, it could not correctly decide, if a *Cipher.init(...)* overload is valid, even if it considered parameters.

5.4.4 Tikanga

Tikanga [WZ11] builds on Jadet. It extends the simple call-order properties to general Computation Tree Logic formulas on object usages. Specifically, it uses formulas to require a call, to require two calls in order, and to require a call after another. It uses model checking to find those formulas from the code. Then, it applies Formal Concept Analysis [GW99] to obtain patterns and violations. As it is based on Jadet, and does not extend the range of Jadet's detectable violations, it finds the same violations and suffers from the same problems. It can find missing *Cipher.init(...)* calls. However, it cannot find insecure configurations of *Cipher*.

5.5 Running the Benchmark

We previously set up MuBench with the new data in Section 5.2 and the new feature to run experiments on specific misuses only in Section 5.3. With this, we can now evaluate four API-misuse detectors on *Cipher*.

To assess the capabilities of DMMC [MM13], GROUMiner [NNP+09], Jadet [WZL07], and Tikanga [WZ11], they have to mine patterns on each project version from the new dataset, and output their findings using these patterns. As detectors output a very large amount of findings, a complete manual review is not feasible. To reduce the output, *MuBench* filters it by potential hits, as described in Section 5.1. We then manually review the filtered findings for each detector.

We run the experiment on the *Cipher* dataset, which consists of 15 misuses, spread over 10 projects. Our results are, that none of the four approaches had potential hits. We discuss why that might be the case in Section 5.6.

5.6 Why API-Misuse Detection Fails on *Cipher*

Concerning the capabilities of API-misuse detectors on *Cipher*, in Section 5.5, we conclude that current API-misuse detectors do not find *Cipher* misuses. In this section, we investigate why that is the case to find possible improvements to the approaches in Section 5.7.

```
1  import java.security.*;
2  import javax.crypto.*;
3  import javax.crypto.spec.IvParameterSpec;
4
5  public class AESEncryption {
6    Cipher cipher;
7
8    public AESEncryption() {
9      cipher = Cipher.getInstance("AES/CTR/NoPadding");
10   }
11
12   public byte[] encrypt(byte[] dataToEncrypt, SecretKey key) {
13     cipher.init(Cipher.ENCRYPT_MODE, key);
14     return cipher.doFinal(dataToEncrypt);
15   }
16 }
```

Figure 5.4: A typical multi-procedural usage of *Cipher*.

5.6.1 Configurations

The API-misuse detectors we evaluated do not consider parameters in their models. However, in the case of *Cipher*, this is a very critical part of the usage, as the parameter of *Cipher.getInstance(...)* can completely change the correct usage patterns for a *Cipher* instance. Since this parameter essentially decides on the concrete type returned, API-misuse detectors must be able to consider it. Even from a very basic point of view, we can already find most violations just by analyzing the parameter itself, since it contains most of the configuration. If this was available in the model, finding insecure configurations would be only a matter of having a set of patterns with secure parameters.

5.6.2 Inter-Procedural Usages

Most API-misuse detectors only consider the scope of a single method. This greatly reduces the complexity and runtime of misuse detection. However, they can not find inter-procedural misuses. For this reason, we implement a Boa program to find *Cipher* fields. We suspect that this is a good indicator to inter-procedural usages. In fact, in our reviews we found *Cipher* very commonly set in a field by the constructor and used later by separate encryption and decryption methods. In Figure 5.4, we show a typical *Cipher* usage we encountered in our manual reviews. Note that this usage also shows multiple misuses, part of which come from the multi-procedural usage. *Cipher.init(...)* does not promise to reset the *Cipher* instance. Hence, it could leave some information from previous encryptions. This can break the subsequent encryption, and lead to data losses. Also, if there where other methods which share the same *Cipher* instance, and initialize it with other values, the outputs could depend on the sequence of these methods. Finally, we omitted all exception handling. In this case, we knowingly did this to minimize the example, but in our manual review, we find that only 9 out of the 26 projects we reviewed exhaustively handled exceptions.

5.6.3 Mining Bad Patterns

The API-misuse detectors we evaluate use a majority rule for their pattern mining, meaning they assume, that, with enough usages, a correct pattern can be found. Our results in Chapter 4 suggest that this is not the case for *Cipher*. We find very frequent patterns in misuses, such as the often used *DES* algorithm. This leads to bad patterns.

5.6.4 Provider Ambiguity

As *Cipher* uses a provider pattern, API-misuse detectors cannot analyze the provider's source code for their approaches. This severely reduces the information available to the detector, since it does not allow the detector to do any analysis on the API's source code. The provider pattern also hides the actual classes from the detectors. Since most detectors build a set of patterns per class, they mix up the patterns for different classes. For example, detectors often cannot differentiate between usages of *AES*, and usages of *DES*, since the patterns for both are stored as patterns for *Cipher*.

5.7 How to Improve API-Misuse Detection

In this chapter, we discuss what future work can do to improve API-misuse detection on *Cipher* and comparable APIs. As shown in Section 5.6, `DMMC` [MM13], `GROUMiner` [NNP+09], `Jadet` [WZL07], and `Tikanga` [WZ11] do not work on *Cipher*, since misuses are much too frequent, and, therefore, the pattern mining may not find any patterns, or even bad patterns. Hence, API-misuse detectors need to find other ways of creating patterns for APIs.

5.7.1 Formal Specifications

A way to obtain correct patterns for APIs such as *Cipher* could be to create a formal specification for API usage. This could be given to an API-misuse detector to produce a complete and correct set of patterns, and would be especially helpful in often misused APIs, as the mining approach generally does not work on them. Alas, this is generally not feasible for all APIs, since creating such a formal usage guide takes a lot of time and is very hard to maintain. Hence, we cannot expect this kind of support for most APIs.

5.7.2 Inter-Procedural Patterns

Another approach to improve patterns may be to support inter-procedural patterns. We find that most pattern models only support usages in a method scope. Since many *Cipher* usages are, as seen in Section 5.6, inter-procedural we suspect that extending patterns to support these usages would improve misuse detection for *Cipher*. However, this significantly increases their complexity.

5.7.3 New Pattern-Mining Approaches

Cipher usages are, in fact, often discussed on sources such as *Stack Overflow* [SE]. During our research, we frequently found that the accepted answers on Stack Overflow use secure configurations for *Cipher*, and explicitly mention that using defaults is a bad practice [O'P], as described in Chapter 2. Using code

from accepted answers of Stack Overflow questions might prove more consistent than mining actual projects. Future work can evaluate, if the higher level of security of snippets we observed in accepted answers in Stack Overflow is indeed relevant, and implement a mining approach that uses these snippets as its source.

If such a mining approach is implemented, it should be compared to existing approaches, for example, mining project source code. To accurately assess the capabilities of these mining approaches, a new experiment for MuBench would proof useful. Such an experiment requires API-misuse detectors to provide access to the mined patterns, hence, future work would have to add this functionality for the existing approaches. Since the detectors may produce too many patterns to review manually, we also need a way to compare the mined patterns to the correct patterns provided by MuBench to filter the potentially correct patterns before a manual review. To make sure the experiment results are valid, an initial validation of the hand-crafted patterns is required, i.e. checking, if the approach indeed produces the correct output when provided with the correct patterns. On the side, such a form of validation supports MuBench's current *Experiment 1*, which simulates a perfect mining stage, and therefore highly depends on assuming that correct patterns are mined. For the new experiment, we still require manual reviews, since we expect that the pattern validation cannot be fully automatized. With this experiment, we can estimate the capabilities of different pattern-mining approaches in isolation, i.e. independent of the misuse-detection stage.

Conclusion

Using the data from Chapter 4, we implement a new dataset in MuBench in Section 5.2. To enable MuBench to run only on our new dataset, we add a new command-line option and specify datasets in a separate configuration file.

Using the functionality we added, we run the detectors `DMMC` [MM13], `GROUMiner` [NNP+09], `Jadet` [WZL07], and `Tikanga` [WZ11] on our *Cipher*-misuse dataset. We find that none of these approaches were able to detect the misuses.

In Section 5.6, we discuss the reasons why API-misuse detection may fail to find *Cipher* misuses. We notice that the detectors do not consider parameter values, which is critical in case of *Cipher*, since its usage entirely hinges on the actual parameter of *Cipher.getInstance(...)*. We also find that the patterns are built on a per-method basis. As most *Cipher* usages split the instance initialization and the creation of ciphertext into different procedures, the detectors often do not find relevant patterns in these cases. Another issue in the pattern mining stage is that the approaches rely on frequency, which, in the case of *Cipher*, does not equate to correctness, as we have seen from our discussion in Section 4.4. Lastly, we also note that detectors build their models per type. Since the actual type of a *Cipher* instance can only be determined during runtime, as it depends on the provider, detectors cannot differentiate which pattern to apply.

In Section 5.7, we suggest what future work can do to improve API-misuse detection on *Cipher*. As all of the weaknesses of the API-misuse detectors we identify in Section 5.6 relate to their pattern-model approaches, we discuss possible improvements in this area. Our first realization is that their current pattern-mining approaches do not find the correct patterns for *Cipher*. We contemplate using formal API specifications to provide detectors with correct and exhaustive patterns, but conclude that this approach is not generally feasible for all APIs, though it can be done in specific cases, as it is currently worked on for *Cipher* by Krüger [Krü]. We suggest evaluation of other sources than random projects, and use Stack Overflow [SE] as an example for an alternative source of correct usages. To compare these approaches, we propose a new experiment for MuBench, which aims to evaluate the precision of pattern-mining

approaches, by comparing the mined patterns to the correct hand-crafted patterns MuBench already provides.

To summarize our findings, API-misuse detectors can probably not solve the problem completely. *Cipher* uses a very high abstraction to provide a lot of flexibility. This comes at the cost of usability, as discussed in Section 4.4. There is a large amount of possible configurations, which are correct API usages considering they produce no error and hold to their specification, but are at the same time considered misuses, which we describe as *API misconceptions* in Section 4.4, since they create security vulnerabilities. Hence, we feel even if tools may improve to find more complex misuses, which would increase the overall code correctness, the main reason why *Cipher* in specific is misused as often is the lack of knowledge about cryptographic concepts. However, we think that API-misuse detectors could very well warn developers in many cases, which would prove helpful, considering the main problem concerning *Cipher* is the inconspicuousness of security vulnerabilities.

6 Summary

As an introduction to the topic, we start in Chapter 2 by introducing *Cipher*, the encryption and decryption API of the *Java Cryptography Extensions (JCE)*. We describe the provider-based architecture, and some of its inherent problems. In Section 3.1, we check, where *Cipher* misuses fit in the API-Misuse Classification (MuC) [ANN+16], to see, if they are comparable to common API misuses. We show that, according to the study of Monperrus et al. [METM12], the directives applicable for *Cipher* are neither rare nor very specific. As we also mapped these directives to suitable classes of the MuC, future work could use this to evaluate, if the respective violations are frequent as well. We suspect that there is no inherent correlation between directive and violation frequency, but a closer comparison could surface some weak areas of either API documentation, or API-misuse detection.

As we need to build a *Cipher* dataset for MuBench [ANN+16], an API-misuse dataset and API-misuse detector benchmarking tool, we use Boa [DNRN13] in Section 4.3 to mine GitHub for *Cipher* usages. During our manual review of 26 projects, we find 25 misuses distributed among 18 projects. We analyze these misuses to see how *Cipher* is misused and what the prevalent problems of the API are in Section 4.4. We notice that the user's constraints are very important when considering usages of APIs, which is especially critical for *Cipher*. In this case, security is a major issue for users, but very badly represented in API documentation. To provide a wide array of functionality, *Cipher* has very much weakened constraints on API usage. This intrinsically hurts its usability, as users can easily come to misconceptions about the API, such as an inherent trust in its security. We introduce *API misconceptions* as an extension to the current definition of API misuses, where API misconceptions are usages, where the user violates one of his own constraints. In the case of *Cipher*, the constraint most likely to be violated is security. However, in general, there are other API misconceptions, such as low performance. As these constraints tend to be hard to validate, their violations often remain unnoticed. We suspect that API-misuse detectors can help the issue, though we cannot estimate how much.

To see how current API-misuse detection approaches perform on *Cipher*, we use MuBench [ANN+16], to evaluate DMMC [MM13], GROUMiner [NNP+09], Jadet [WZL07], and Tikanga [WZ11], four state-of-the-art API-misuse detectors. In Section 5.3, using the dataset from Chapter 4, we extend MuBench, and subsequently evaluate the detectors. Our evaluation results show that these approaches do not find the misuses in our dataset. In Section 5.6, we discuss, why the approaches do not work on *Cipher*. We find that they fail to find the correct patterns for *Cipher* usages.

Finally, in Section 5.7, we discuss what future work can do to improve API-misuse detection on APIs such as *Cipher*. We ponder using formal API specifications to generate patterns, but argue that this is not generally feasible for APIs. However, it may be very helpful for specific APIs, and is, in case of *Cipher*, currently worked on by Krüger [Krü]. However, for APIs without such specifications, we suggest to find alternative mining approaches, such as mining Stack Overflow [SE]. To compare these mining approaches, future work could also add a new experiment to MuBench, where the mining stage is evaluated in isolation. We also find that including parameter values into the pattern model is invaluable for detecting *Cipher* misuses.

We find that even though the approaches we evaluated did not find *Cipher* misuses, API-misuse detection could still prove promising to help developers using these APIs. However, as can be seen from our discussion, there are still a lot of improvements to be made to API-misuse detection.

List of Figures

References

[ANN+16] Sven Amann, Sarah Nadi, Hoan A. Nguyen, Tien N. Nguyen, and Mira Mezini. Mubench: A benchmark for api-misuse detectors. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 464–467. ACM, New York, NY, USA, 2016. ISBN 978-1-4503-4186-8. URL http://doi.acm.org/10.1145/2901739.2903506.

[BC] Inc. Bouncy Castle. Bouncy castle. URL https://www.bouncycastle.org/java.html.

[DiB15] Chris DiBona. Bidding farewell to google code, 2015. URL https://opensource.googleblog.com/2015/03/farewell-to-google-code.html.

[DNRN13] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE'13, pages 422–431, May 2013.

[EBFK13] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 73–84. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-2477-9. URL http://doi.acm.org/10.1145/2508859.2516693.

[Foua] The Apache Software Foundation. Apache ant. URL http://ant.apache.org/.

[Foub] The Apache Software Foundation. Apache maven project. URL https://maven.apache.org/.

[Goo] Inc. Google. Android developers cipher documentation. URL https://developer.android.com/reference/javax/crypto/Cipher.html.

[Gra] Inc. Gradle. Gradle build tool. URL https://gradle.org/.

[GW99] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, Berlin/Heidelberg, 1999.

[Krü] Stefan Krüger. Java cryptography extension specification. URL https://www.hni.uni-paderborn.de/swt/mitarbeiter/1456475238/.

[Lin07] Christian Lindig. Mining patterns and violations using concept analysis. Technical report, Universität des Saarlandes, Saarbrücken, Germany, jun 2007.

[LZ05] Zhenmin Li and Yuanyuan Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 306–315. ACM, New York, NY, USA, 2005. ISBN 1-59593-014-0. URL http://doi.acm.org/10.1145/1081706.1081755.

[METM12] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What Should Developers Be Aware Of? An Empirical Study on the Directives of API Documentation. *Empirical Software Engineering*, 17(6):703–737, 2012. ISSN 1382-3256. URL http://www.monperrus.net/martin/An-Empirical-Study-On-the-Directives-of-API-Documentation.pdf.

[MM13] Martin Monperrus and Mira Mezini. Detecting missing method calls as violations of the majority rule. *ACM Trans. Softw. Eng. Methodol.*, 22(1):7:1–7:25, March 2013. ISSN 1049-331X. URL http://doi.acm.org/10.1145/2430536.2430541.

[NKMB16] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 935–946. ACM, New York, NY, USA, 2016. ISBN 978-1-4503-3900-1. URL http://doi.acm.org/10.1145/2884781.2884790.

[NNP+09] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-001-2. URL http://doi.acm.org/10.1145/1595696.1595767.

[NPVN15] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen. Recommending api usages for mobile apps with hidden markov model. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 795–800, Nov 2015.

[O'P] Zim-Zam O'Pootertoot. Stackoverflow answer correcting default usage. URL http://stackoverflow.com/a/15926867.

[Oraa] Oracle. Javadoc tool. URL http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html.

[Orab] Oracle. Sunjce. URL http://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html#SunJCEProvider.

[SE] Inc. Stack Exchange. Stack overflow. URL http://stackoverflow.com/.

[TX09a] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 283–294, Nov 2009. ISSN 1938-4300.

[TX09b] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *2009 IEEE 31st International Conference on Software Engineering*, pages 496–506, May 2009. ISSN 0270-5257.

[WZ11] Andrzej Wasylkowski and Andreas Zeller. Mining temporal specifications from object usage. *Automated Software Engineering*, 18(3):263–292, 2011. ISSN 1573-7535. URL http://dx.doi.org/10.1007/s10515-011-0084-1.

[WZL07] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 35–44. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-811-4. URL http://doi.acm.org/10.1145/1287624.1287632.

[Zie15] Henning Ziegler. Analyse der verwendung von kryptographie-apis in java-basierten anwendungen. Master's thesis, Universität Bremen, 11 2015.