Dealing with Variability in API Misuse Specification

(Artifact Package)

Rodrigo Bonifácio, Stefan Krüger, Krishna Narasimhan, Eric Bodden, and Mira Mezini

March, 2021

Abstract

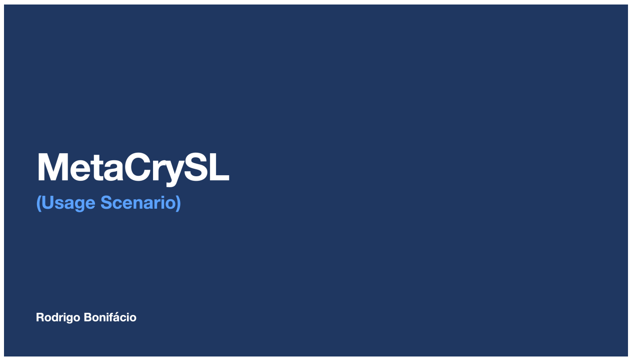
APIs are the primary mechanism for developers to gain access to externally defined services and tools. However, previous research has revealed API misuses that violate the contract of APIs to be prevalent. Such misuses can have harmful consequences, especially in the context of cryptographic libraries. Various API-misuse detectors have been proposed to address this issue—including CogniCrypt, one of the most versatile of such detectors and that uses a language (CrySL) to specify cryptographic API usage contracts. Nonetheless, existing approaches to detect API misuse had not been designed for systematic reuse, ignoring the fact that different versions of a library, different versions of a platform, and different recommendations/guidelines might introduce variability in the correct usage of an API. Yet, little is known about how such variability impacts the specification of the correct API usage. This paper investigates this question by analyzing the impact of various sources of variability on widely used Java cryptographic libraries (including JCA/JCE, Bouncy Castle, and Google Tink). The results of our investigation show that sources of variability like new versions of the API and security standards significantly impact the specifications. We then use the insights gained from our investigation to motivate an extension to the CrySL language (named MetaCrySL), which builds on meta-programming concepts. We evaluate MetaCrySL by specifying usage rules for a family of Android versions and illustrate that MetaCrySL can model all forms of variability we identified and drastically reduce the size of a family of specifications for the correct usage of cryptographic APIs. **In this artifact package**, we present the tools, datasets, and scripts used in the research.

(1) MetaCrySL implementations

We have two implementations of MetaCrySL: the first, based on Rascal-MPL, is discussed in the paper; the second, based on XText, is the most up-to-date implementation.

- MetaCrySL in Rascal
- MetaCrySL in XText

Detailed instructions about how to set up and use these implementations are available in the respective repositories. The following video explains how to use the MetaCrySL Rascal version.



MetaCrySL

(Usage Scenario)

Rodrigo Bonifácio

(2) Domain Analysis

To better understand the impacts of variability on API misuse specification, we conducted a domain analysis that sought to understand reuse opportunities across Crypto-API- usage specifications, considering different libraries and their different versions, different cryptographic primitives, and different cryptographic standards.

As such, we answer the following research questions.

- (RQ1) How do different APIs and their implementations vary the specifications of the correct usage of cryptographic primitives?
- (RQ2) How do existing cryptographic standards vary the notion of secure or compliant use of cryptographic libraries?
- (RQ3) How does the evolution of a cryptographic library vary its correct usage over time?

To answer the first research question we started reading the official documentation, books, tutorials in gray literature, code examples, and test cases of the following libraries.

- Java JCA / JCE
- Java Bouncy Castle
- Java Google Tink
- C/C++ WolfCrypt
- C/C++ OpenSSL

We also contribute by writing CrySL specifications for several primitives of the Bouncy Castle and Google Tink libraries. These specifications could be found in the CrySL Rules repository. To answer the second research question, we mined the documentation of three cryptographic standards:

- NIST FIPS
- BSI
- Ecrypt D5.4

and updated an initial set of CrySL specifications tailored for each one of these cryptographic standards. Afterwards, we generated complete sets of CrySL specifications using MetaCrySL. To anwswer the third research question we mined the evolution of Java cryptographic libraries (JCA/JCE, Bouncy Castle, and Google Tink).

Cryptographic API Evolution and Breaking Changes

- API Evolution tool
- Breaking Changes tool

  We implemented a slight extension to the APIDiff library that computes the breaking changes between two revisions. This additional feature could be used according to the following test case.

```
@Test
public void testMethodBreakingChanges() {
  try {
    // commit id for revision r1rv60 of Bouncy Castle
    String r1rv60 = "52b0902592e770b8116f80f2eab7a4048b589d7d";
    // commit id for revision r1rv59 of Bouncy Castle
    String r1rv59 = "6de1c17dda8ffdb19431ffcadbce1836867a27a9";

    String out = getClass().getResource("/").getFile();

    APIDiff diff = new APIDiff("bc", "https://github.com/bcgit/bc-java.git");

    diff.setPath(out);

    Result res = diff.detectChangeBetweenRevisions(r1rv60, r1rv59, Classifier.API);

    long methodBreakingChanges = res.getChangeMethod().stream()
                                     .filter(c -> c.isBreakingChange())
                                     .count();

    Assert.assertEquals(40, methodBreakingChanges); // expecting 40 methodBreakingChanges

  }
  catch(Exception ex) {
     Assert.fail();
  }
}
```

We used this program to compute the breaking changes from Bouncy Castle and Google Tink. Regarding JCA / JCE, we follow a manual approach of reading the JavaDoc of the API classes among its different revisions.

Datasets and Scripts

- Summary of API evolution
  - releases
- Summary of Breaking Changes
  - Bouncy Castle dataset
  - Google Tink dataset
- Scripts:
  - RMD
  - HTML

(3) MetaCrySL empirical assessment

We evaluated MetaCrySL regarding two dimensions: (a) Compactness (how many lines of specification we can save using MetaCrySL and how much duplication of specifications we can save using MetaCrySL) and (b) Correctness (observing the implications of using the outcomes of MetaCrySL to mine the incorrect usage of crypto API).

In this study we used MetaCrySL to specify different sets of CrySL specifications for the Android platform. These specifications address all JCA/JCE cryptographic primitives, the guidelines from three cryptographic **recommendations** (from the BSI standard, from the CogniCrypt project, and from the Google Android Cryptographic documentation), and three ranges of the Android platform versions (01 – 08, 01 – 16, 01 – 28).

The following table shows the nine configurations we derive from these MetaCrySL specifications.

Config. Id	Primitives	Android Platform Version	Crypto Standard
C01	All primitives	01 – 08	Android Base recommendations
C02	All primitives	01 – 16	Android Base recommendations
C03	All primitives	01 – 28	Android Base recommendations
C04	All primitives	01 – 08	Android BSI Standard recommendations
C05	All primitives	01 – 16	Android BSI Standard recommendations
C06	All primitives	01 – 28	Android BSI Standard recommendations
C07	All primitives	01 – 08	Android CogniCrypt recommendations
C08	All primitives	01 – 16	Android CogniCrypt recommendations
C09	All primitives	01 – 28	Android CogniCrypt recommendations

MetaCrySL Artifacts used in this study

- MetaCrySL JCA/JCE specifications
- MetaCrySL Android Base (refinements and configurations)
- MetaCrySL Android BSI (refinements and configurations)
- MetaCrySL Android CogniCrypt (refinements and configurations)

Generated CrySL rules

- Android Base (all versions)
- Android BSI (all versions)
- Android CogniCrypt (all versions)

Binary Version of CrySL Rules

We used the CogniCrypt Eclipse plugin to compile the generated CrySL rules. The compiled version of the CrySL rules are used as input to the CogniCrypt SAST tool.

- Android Base (all versions)
- Android BSI (all versions)
- Android CogniCrypt (all versions)

Datasets

- List of apps used in the study
- Violations
  - version android-0108
  - version android-0116
  - version android-25plus
  - version android-bsi-0108
  - version android-bsi-0116
  - version android-bsi-25plus
  - version android-cc-0108
  - version android-cc-0116
  - version android-cc-25plus

Scripts

- RMD
- HTML

Live version of this package

An on-line and **live** version of this package is also available at this public repository.