CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs (Artifact)

Stefan Krüger
Paderborn University, Germany
firstname.lastname@uni-padeborn.de

Johannes Späth
Fraunhofer IEM, Germany
johannes.spaeth@iem.fraunhofer.de

Karim Ali
University of Alberta, Canada
karim.ali@ualberta.ca

Eric Bodden
Paderborn University, Germany
firstname.lastname@uni-padeborn.de

Mira Mezini
Technische Universität Darmstadt, Germany
mezini@cs.tu-darmstadt.de

Abstract

In this artefact, we present CrySL, an extensible approach to validating the correct usage of cryptographic APIs. The artefact contains executables for CogniCrypt$_{\text{SAST}}$, the analysis CrySL-based analysis, along with the CrySL rules we used in in the original paper's experiments. We also provide scripts to re-run the experiments. We finally include a tutorial to showcase the CogniCrypt$_{\text{SAST}}$ on a small Java target program.

2012 ACM Subject Classification Security and privacy → Software and application security, Software and its engineering → Software defect analysis, Software and its engineering → Syntax & Semantics

Keywords and phrases cryptography, domain-specific language, static analysis

Digital Object Identifier 10.4230/DARTS.VOL.ISS.ART

Related Conference 32nd European Conference on Object-Oriented Programming (ECOOP 2018), July 19–21, 2018, Amsterdam, Netherlands

1 Scope

The artefact is supposed to support repeatability of the experiments in the original paper on a much smaller scale. In particular, it is designed such that the analysis CogniCrypt$_{\text{SAST}}$ may be applied to some of the apps we used in our evaluation. Lastly, it facilitates running CogniCrypt$_{\text{SAST}}$ on arbitrary Android and Java applications.

2 Content & Usage

The artefact is a docker container that provides the CogniCrypt$_{\text{SAST}}$ analysis, as well as the rule sets used for RQ2 and RQ4. We provide the full analysis including a version specifically built to analyse Android apps as well as the CrySL rules from our evaluation. We also provide a few test apps for the analysis, but have significantly reduced their number compared to the original paper because the full analysis takes several days to run even on a 16 core machine with 64 GB RAM.

To set up the docker container, please first download the file called *crysl-artefact* from the location given in Section 3. The file is a raw docker image that first needs to be imported into the local docker installation before it can be launched. To this end, execute the commands in the directory with the *crysl-artefact* file.

```
docker import crysl-artefact
docker run -ti -v $absolute/Path/on/your/host/system:/home/output
    $hash_of_image /bin/bash
```

The first command imports the image within the file. The *docker run* command launches a container for the image. The *ti* option sets up a shell in the container and automatically connects to it. The -v option creates a shared volume between container and host system. The folder */home/output* has already been set up, but a directory on the host system needs to be selected that should serve as the shared volume (see $absolute/Path/on/your/host/system). The directory is used to store the analysis results, facilitating their inspection from the host system. Following that, one needs to specify the image ID, which one can get by executing `docker images` and then taking the ID of the most recently added image, as well as the initial command */bin/bash* to set up and launch the shell in the container.

When running the docker *run* command, the docker container launches at */*. Navigate to */home*, in which one may find three folders. First, there is the previously discussed *output* folder, next to the folders *JavaAnalysis* and *AndroidAnalysis*. Folder *JavaAnalysis* contains a small Java example for the analysis that serves as a tutorial to the artefact and which we describe further in section 2.1. Lastly, folder *AndroidAnalysis* contains the tools and data to reproduce our results.

2.1 Java Tutorial

In the *JavaAnalysis* directory, one may find several files that all relate to the analysis. First, the *CryptoAnalysis.jar* comprises the CogniCrypt$_{\text{SAST}}$ analysis itself. It further contains the target project *FileEncryptor*. The project implements a simple file encryption, but contains a few bugs CogniCrypt$_{\text{SAST}}$ picks up. Finally, the directory *CryslRules* contains the full Ruleset$_{\text{FULL}}$ rule set, both as binaries and in textual form. The latest version of the rules are available at `https://github.com/CROSSINGTUD/Crypto-API-Rules`. To execute the analysis on the target project, we provide the two scripts *runStdOutAnalysis.sh* and *runFileOutAnalysis.sh*. They can be executed as follows:

```
./runFileOutAnalysis.sh
```

The former prints the analysis report to the console, the latter stores them in a file in */home/output/Javareports* (ergo also on the shared folder of the host system). The report file for the tutorial target project is displayed below. The header of the file lists all involved CrySL rules in case the user wishes to check the rule their program violated. The actual findings are grouped by class and further by method name. Each finding contains a short description of the misuse and displays the statement the misuse was found at in Jimple, the intermediate representation the analysis framework Soot [?] we have built CogniCrypt$_{\text{SAST}}$ on operates on. The former is to help the user figure out quickly what they have done wrong and how to fix it, the latter should support them in finding the affected location easily. Applying this structure, the first finding in the report below can be interpreted as "In method `encrypt` of class `Crypto.Enc`, the parameter first parameter of the call to `Cipher.getInstance()` should not just be *AES* but be extended with one of the elements in the list." We suggest the reader to check out the rules in the docker image or online and either introduce more rule violations to the target program or fix the ones CogniCrypt$_{\text{SAST}}$ finds in it.

```
Ruleset:
        SecretKey
        ...
        SecureRandom
        Cipher
        Signature
        KeyGenerator
        ...
        SecretKeyFactory

Findings in Java Class: Crypto.Enc
  in Method: byte[] encrypt(java.lang.String,javax.crypto.SecretKey)
    "AES" should be any of AES/{CBC, GCM, PCBC, CTR, CTS, CFB, OFB}
      @r3 = staticinvoke <javax.crypto.Cipher: javax.crypto.Cipher
          getInstance(java.lang.String)>("AES")
        Variable r2 of type javax.crypto.SecretKey was not properly
            generatedKey
      @virtualinvoke r3.<javax.crypto.Cipher: void init(int,java.security
          .Key)>(1, r2)
  in Method: java.lang.String decrypt(byte[],javax.crypto.SecretKey)
    "AES" should be any of AES/{CBC, GCM, PCBC, CTR, CTS, CFB, OFB}
      @r3 = staticinvoke <javax.crypto.Cipher: javax.crypto.Cipher
          getInstance(java.lang.String)>("AES")

Findings in Java Class: FileHandler
  in Method: java.lang.String performEncryption(java.lang.String)
    Object of type byte[] was not properly randomized
      @specialinvoke $r4.<javax.crypto.spec.SecretKeySpec: void <init>(
          byte[],java.lang.String)>($r6, "AES")
```

2.2 Experiments

In the *AndroidAnalysis* folder, one can find all files related to reproducing our experiments. In directory *apps*, we provide a few apps along with the artefact in order to facilitate the execution of the analysis. We direct any readers who wish to re-run the full analysis to AndroZoo [?] and Section 8 of our paper in which we outline the selection criteria for the apps. In any case, the folder further contains the rule sets RULESET$_{\text{FULL}}$ in *CogniCryptRules* and RULESET$_{\text{CL}}$ in *CryptoLintRules*, both in their binary and textual form. We used the RULESET$_{\text{FULL}}$ in answering all research questions, the RULESET$_{\text{CL}}$ for RQ4 only. As we analyse Android apps, we require platform files for different versions of the Android SDK in *platforms*. On top of that, we also need the Android-aware variant of COGNICRYPT$_{\text{SAST}}$ *CryptoAnalysis-Android-1.0.0-jar-with-dependencies.jar*. It comes with some wrapper code that deals the Android-specific content of the apk files and uses Flowdroid [?] for call-graph construction. Once that is done, COGNICRYPT$_{\text{SAST}}$ resumes on the remaining Java code. To launch the analysis, execute one of the two *runCogniCryptRulesAnalysis.sh* or *runCryptoLintRulesAnalysis.sh* scripts, depending on which rule set you want applied. Note that we limit the execution time of the analysis to ten minutes by means of *timeout*. We opted for this solution as the execution time fluctuated heavily between five and 25 minutes on our different testing machines.

The analysis stores its results in */home/output/Androidreports*. For each app, a report file following the above described structure is created. Additionally, the analysis summarizes the results in a *.csv* file.

3 Getting the artefact

You may download the artefact at `https://uni-paderborn.sciebo.de/s/uLtxYDv3Aafob2L`.

4 Tested platforms

The artefact has been tested with Docker for Windows 10.

5 License

The whole artefact licensed under Eclipse Public License (EPL) Version 2.0 (`https://www.eclipse.org/legal/epl-2.0/`). This does not hold for the apps we provide along with the artefact. They remain licensed under their own license.

6 MD5 sum of the artefact

b6c347f79bd437978b1cc8d0c018ba16

7 Size

2.0 Gb