MASC: A Tool for Mutation-Based Evaluation of Static Crypto-API Misuse Detectors

Amit Seal Ami
aami@wm.edu
Computer Science Department, William & Mary Williamsburg, Virginia, USA

Syed Yusuf Ahmed
radowan@umich.edu
Institute for Information Technology, University of Dhaka Dhaka, Bangladesh

Nathan Cooper
kkaflw@wm.edu
Computer Science Department, William & Mary Williamsburg, Virginia, USA

Kevin Moran
kpmoran@ucf.edu
Department of Computer Science, University of Central Florida Orlando, Florida, USA

Denys Poshyvanyk
denis@cs.wm.edu
Computer Science Department, William & Mary Williamsburg, Virginia, USA

Adwait Nadkarni
apnadkarni@wm.edu
Computer Science Department, William & Mary Williamsburg, Virginia, USA

ABSTRACT

While software engineers are optimistically adopting crypto-API misuse detectors (or crypto-detectors) in their software development cycles, this momentum must be accompanied by a rigorous understanding of crypto-detectors’ effectiveness at finding crypto-API misuses in practice. This demo paper presents the technical API misuses in practice.

KEYWORDS

Security and privacy, Software security engineering.

CCS CONCEPTS

- Security and privacy → Software security engineering.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ACM Reference Format:

1 INTRODUCTION

Software engineers have been relying on crypto-detectors for decades to ensure the correct use of cryptographic APIs in the software and services they create, develop, and maintain [4]. Such crypto-detectors are ubiquitous in software engineering, as they are integrated into IDEs (e.g., CogniCrypt plugin for Eclipse IDE [5]), testing suite of organizations such as Oracle Corporation [6, 12], and for Continuous Integration/Continuous Deployment (CI/CD) pipelines [9, 13]. In addition, hosting providers such as GitHub are formally provisioning such crypto-detectors e.g., GitHub Code Scan Initiative [7]. In other words, the security of software and services are increasingly becoming more reliant on crypto-detectors. However, we have been relying on manually-curated benchmarks for evaluating the performance of crypto-detectors, such benchmarks are known to be incomplete, incorrect, and impractical to maintain [11]. Therefore, determining the effectiveness of crypto-detectors from a security-focused perspective requires a reliable and evolving evaluation technique that can scale with the volume and diversity of crypto-API and the different patterns of misuse.

We contextualized mutation testing techniques to create the Mutation Analysis for evaluating Static Crypto-API misuse detectors (MASC) framework. In our original, prototype implementation of MASC [3], it internally leveraged 12 generalizable, usage-based mutation operators to instantiate mutations of crypto-API misuse cases for Java. The mutation operators were designed based on the design principles of Java Cryptographic Architecture (JCA) [8] and a threat model that consisted of users of varying skills and intentions (Section 4.1). MASC injects these mutated misuse cases in Java or Android-based apps at three mutation scopes (injection sites),

"..."
We create and introduce an open-sourced, web-based front-end of MASC without diving deep into the existing code base (Tool and Data Availability: The prototype implementation of the MASC framework, scripts and results of evaluating crypto-detectors, as described in the original paper [3], are available in the MASC Artifact [1]. Furthermore, the codebase of actively maintained, mature implementation of MASC is available separately with extensive documentation and examples [2].

2 OVERVIEW OF MASC

Overall, MASC works by (1) mutating a base crypto API misuse case to create mutated crypto-API instantiations or mutated misuse case, (2) seeding or injecting the mutated misuse case in source code, (3) analyzing both unmutated and mutated source code using a target crypto-detector, and (4) comparing the outputs of crypto-detector applied on both base misuse case and mutated misuse case to identify undetected (not killed) mutated misuse case. The overview of this process is shown in Figure 1.

Conceptually, MASC contextualizes the traditional mutation testing techniques of SE domain for the evaluation of crypto-detectors, while introducing crypto-API misuse mutation operators that instantiates variants or expressions of crypto-API misuse. To elaborate, while mutation operators from the traditional, SE mutation testing are used to describe operations that either add, modify, or remove existing source code statement(s), in the context of MASC, crypto-API mutation operators create expressive instances of crypto-API misuse independent of any source code or application. As shown in Listing 1, statement marked //1 is the base misuse case, whereas statements //2 – //5 are the mutated crypto-misuse cases instantiated by several mutation operators of MASC. We provide the design considerations and implementation details of MASC’s mutation operators in Sec. 4.1. These mutated misuse instances are then “injected” or “seeded” in source code, where the injection site depends on the mutation scopes of MASC, which we detail in Sec. 4.2.

3 DESIGN GOALS

We considered several goals while designing MASC, while leaning on the experience we gained from the original version. Diversity of Crypto-APIs (DG1): Effectively evaluating crypto-detectors requires considering misuse cases of existing crypto-APIs, which is challenging as crypto-APIs are as vast as the primitives they enable.
To address this, the crypto-API mutation operators need to be decoupled from the crypto-APIs. Such implementation would mean that even in the case when new crypto APIs are introduced, MASC can still create mutated misuse cases as long as the new crypto-APIs follow existing design principles.

Open to Extension (DG2): While both original and current implementations of MASC come with 12 generalizable mutation operators, these represent a subset of different expressions of misuse cases. Hence, MASC should be open to extension by stakeholders so that they can create their own mutation operators that can be easily plugged-in to MASC, without needing to modify MASC.

Ease of Evaluating Crypto-detectors (DG3): While the original, semi-automated implementation of MASC required manual evaluating the target crypto-detector, such heavy-lifting manual effort can not be simply expected from end-users. Part of this manual effort was unavoidable due to the unique, varied outputs produced by crypto-detectors. However, with the recent focus on using crypto-detectors with CI/CD pipelines and the introduction of the de-facto SARIF [10] formatted outputs, it would become possible to not only automate the entire evaluation process, but also make it customizable.

Adapting to Users (DG4): Finally, MASC should be created in such a way that it is usable by users of varying skills and in different environments. For instance, it should be usable as a stand-alone binary in a windowless server environment as a component, and as a front-end based software that can leverage the binary of itself.

4 IMPLEMENTATION OF MASC

To satisfy the design goals (DG1–DG4), we implemented MASC (11K+ effective Java source line of code) following single-responsibility principle across modules, classes, and functions. Note that while current implementation of MASC inherits the mutation scopes of the original implementation with internal structural changes, the bulk of the changes with new features in the current implementation of MASC are based on the Main Scope. Therefore, we describe the implementation details of MASC with a focus on Main Scope in the context of the design goals and provide an overview of the architecture in Figure 2.

Configuration Manager: To make MASC as flexible as possible, we decoupled the crypto-API specific parameters from the internal structure of MASC. As a result, user can specify any crypto-API along with its necessary parameters through an external configuration file defining the base crypto-API misuse case. The configuration file follows a key-value format, as shown in Listing 2. Additionally, user can specify the mutation operators and scope to be used, along with other configuration values, thus satisfying DG1.

Figure 2: Architecture Overview of the Main Scope of MASC

<table>
<thead>
<tr>
<th>Configuration Manager</th>
<th>Automated Evaluation Module</th>
<th>Mutation Operator Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract and provide parameters</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Listing 2: Example configuration file for MASC

```
scope = main
type = StringOperator
outputDir = app/outputs
apiName = javax.crypto.Cipher
# Method call from crypto-API
invocation = getInstance
# Secure parameter to use with crypto-API
secureParam = AES/ECB/NoPadding
# Insecure parameter to use with crypto-API
insecureParam = AES
# Noise value used with mutation
noise = 0
# Variable, class name used to create necessary structures
variableName = cryptoVariable
className = CryptoTest
# Name of the app for similarity-scope specific mutation
appName = <name of the app>
```

Mutation Operator Module: MASC analyzes the specified crypto-API and uses the values specified by the user (e.g., secure, and insecure parameters to be used with the API) for creating mutated crypto-API misuse instances. Internally, the decoupling of crypto-APIs from MASC is made possible through the use of Java Reflection based API analysis and Java Source Generation using the Java Poetry Library (DG1). While both the original and current implementation of MASC comes with several generalizable mutation operators, the current implementation of MASC includes an additional plug-in structure that facilitates creating custom mutation operators and custom key-value pairs for the configuration file. Both of these can be done externally, i.e., no modification to source code of MASC is necessary (DG2). We provide additional details about MASC’s mutation operators in Section 4.1.

Automated Evaluation Module: The current implementation of MASC leverages the SARIF formatted output to automate evaluation of crypto-detectors. To make end-to-end analysis automated, MASC’s can be configured to use crypto-detector specific commands, such as e.g., compiling a mutated source code for analysis, evaluation stop conditions, command for running crypto-detector, output directory, and more (DG3–DG4).

Furthermore, MASC is implemented to produce verbose logs. With the combination of flexible configuration, it is therefore possible to use the stand-alone binary MASC jar file as a module of another software. As a proof of concept, we implemented MASC Web, a python-django based front-end that offers all the functionalities of the MASC (Usage details in Section 5) that uses the binary jar of MASC as a module (DG4).

4.1 Mutation Operators

We designed generalizable mutation operators by examining the Java Cryptographic Architecture (JCA) documentation. We identified two common patterns of crypto-API invocation as follows: (i) restrictive, where a developer is expected to only instantiate certain crypto-API objects by providing values from a pre-defined set, e.g., Cipher, and (ii) flexible, where the developers implement the behavior, e.g., HostnameVerifier. While defining mutation operators of these two distinct patterns, we assumed a threat model consisting of the following types of adversaries:

Benign developer, accidental misuse (T1): A benign developer who accidentally misuses crypto-API, but attempts to address such vulnerabilities using a crypto-detector.
We implemented 6 mutation operators for restrictive crypto-APIs. Similarly, for the flexible APIs, we implemented mutation operators based on object-oriented programming concepts:

- **Method overriding** is used to create mutations that contain ineffective security exception statements, irrelevant loops, and/or ineffective security sensitive return value,
- **Class extension** is used for implementing or inheriting parent crypto-API interface or abstract classes respectively, and
- **Object Instantiation** is for creating anonymous inner class object from the implemented or inherited classes of crypto-APIs.

We created 6 more conceptual mutation operators based on flexible crypto-APIs. An example of flexible mutant is shown in Listing 3.

### 4.2 Mutation Scopes

To emulate vulnerable crypto-API misuse placement by benign and evasive developers, we designed three mutation scopes to be used with MASC:

- **Main Scope** represents the simplest scope, where it seeds mutants at the beginning of the main method of a simple Java or Android template app, ensuring reachability.
- **Similarity Scope** seeds mutants in the source code of an input application where a similar crypto-API is found. Note that it does not modify the existing crypto-API, and only appends the said mutant misuse case
- **Exhaustive Scope** seeds mutants at all syntactically possible locations in the target app, such as class definition, conditional segments, method bodies and anonymous inner class object declarations. This helps evaluate the reachability of the target crypto-detector.

### 5 USING MASC

As described previously, MASC has both command line interface and web-based front-end (MASC Web, shown in Figure 3). MASC CLI can be executed by providing a configuration file e.g., Cipher.properties using the command shown in Listing 4. Similarly, using the MASC Web, users can do the following, labeled as per Figure 3:

1. Experiment and learn about crypto-API misuse using MASC Lab,


Received 2023-05-11; accepted 2023-07-20