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 Mahmud Redoy*
bsse1013@iit.du.ac.bd
bsse1002@iit.du.ac.bd
Institute for Information Technology,
University of Dhaka
Dhaka, Bangladesh

Nathan Cooper
Kaushal Kafle
nacooper01@wm.edu
kkafle@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
denys@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 details and usage scenarios of our tool, namely **M**utation **A**nalysis for evaluating **S**tatic **C**rypto-API misuse detectors (MASC). We developed 12 generalizable, usage based mutation operators and three mutation scopes, namely *Main Scope*, *Similarity Scope*, and *Exhaustive Scope*, which can be used to expressively instantiate compilable variants of the crypto-API misuse cases. Using MASC, we evaluated nine major crypto-detectors, and discovered 19 unique, undocumented flaws. We designed MASC to be *configurable* and *user-friendly*; a user can configure the parameters to change the nature of generated mutations. Furthermore, MASC comes with both Command Line Interface and Web-based front-end, making it practical for users of different levels of expertise.
Code: https://github.com/Secure-Platforms-Lab-W-M/MASC

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Crypto-API, static analysis, crypto-API misuse detector, mutation testing, mutation-based evaluation, security, software-engineering

*These authors contributed equally to this paper

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3613099

ACM Reference Format:
Amit Seal Ami, Syed Yusuf Ahmed, Radowan Mahmud Redoy, Nathan Cooper, Kaushal Kafle, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. 2023. MASC: A Tool for Mutation-Based Evaluation of Static Crypto-API Misuse Detectors. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3611643.3613099

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 **M**utation **A**nalysis for evaluating **S**tatic **C**rypto-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), namely *Main Scope*, *Similarity Scope*, and *Exhaustive Scope*, thus creating mutated applications that contain crypto-API misuse. We demonstrated the practicality of prototype implementation of MASC by evaluating nine crypto-detectors from industry and academia, and discovered 19 previously undocumented, unknown flaws that compromise the within-scope soundness of crypto-detectors. The full details of MASC's methodology, design considerations, evaluation of crypto-detectors leading to finding novel flaws, practical impact of found flaws in open source applications (therefore the applicability of the mutation operators), and discussion of the findings are available in the original research paper [3].

In this paper, we present a mature implementation of MASC framework with focus on extensibility, ease of use, and maintainability to the stakeholders of crypto-detectors, such as security researchers, developers, and users. To elaborate, because of the newly developed plug-in architecture, MASC users can now create their own mutation operators that can be easily plugged into MASC, without diving deep into the existing code base ($11K$+ source lines of code). Moreover, whereas the original prototype implementation of MASC involved semi-automated evaluation of crypto-detectors, we made MASC's workflow automated by leveraging the *de-facto* SARIF [10] formatted output of crypto-detectors. Furthermore, we have created a web-based front-end of MASC's implementation for the users to reduce the barrier to entry. Finally, we restructured and refactored the open-source codebase of MASC to increase maintainability and extensibility of MASC, which will make future contributions and enhancements easier for both developers and open-source enthusiasts of MASC. With these additions and enhancements, we hope that the current, open-source implementation of MASC will be used in finding flaws in, and thus helping to improve, existing crypto-detectors.

Contribution: We present MASC, a user-friendly framework that leverages mutation-testing techniques for evaluating crypto-detectors, with details of underlying techniques, design considerations, and improvements. The new, key features of MASC are as follows: *Automated Evaluation of Crypto-detectors:* MASC can be used to evaluate crypto-detectors in an end-to-end automated workflow within the Main Scope.

Customizable Evaluation of Crypto-detectors: A user can customize the evaluation of crypto-detectors by specifying the mutation operators for creating crypto-API misuse instances.

Plug-in Architecture for Custom Operators: MASC helps security researchers, developers and users, jump right into evaluating crypto-detectors by creating their own, custom mutation operators that can be directly plugged-in the *Main* Scope, without requiring them to learn and understand about the internal details of MASC.

User-friendly Front-end for End-users: In addition to enhancing the command line interface of the original prototype implementation, we create and introduce an open-sourced, web-based front-end for end-users that can be run locally. The front-end contains an additional *play-test-learn* interface, MASC Lab, where stakeholders can interact with mutation operators and can learn about mutating crypto-API misuse.

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].

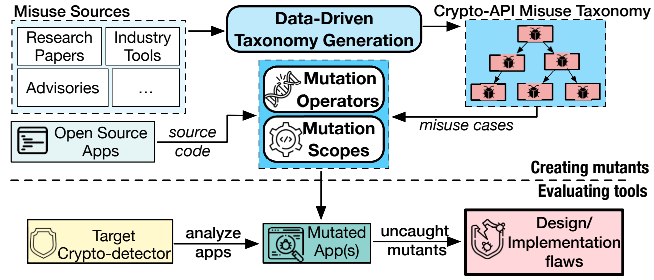

Figure 1: A conceptual overview of the MASC framework.

```
//base crypto API misuse
Cipher.getInstance("DES"); // 1
//mutated misuse instances from several mutation operators of MASC
Cipher.getInstance("des"); // 2
Cipher.getInstance("des".toUppercase()); //3
Cipher.getInstance("DE$S".replace("$","")); // 4
String val = "DES"; Cipher.getInstance(val); // 5
```

Listing 1: Example crypto-API misuse instances created by MASC

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.

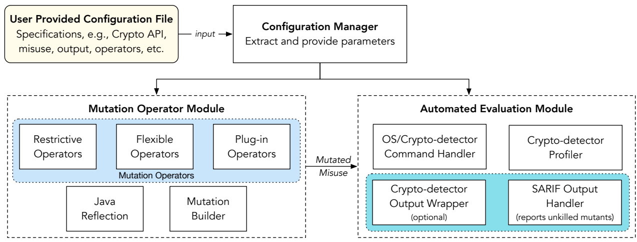

Figure 2: Architecture Overview of the Main Scope of MASC

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**.

```
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/GCM/NoPadding
# insecure parameter to use with crypto-API
insecureParam = AES
# noise value used with mutation
noise = ~
# 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>
```

Listing 2: Example configuration file for MASC

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.

```
interface IHV extends HostnameVerifier{}
new IHV(){
 public boolean verify(String h,SSLSession s)return true;};
```

Listing 3: Flexible crypto-API based misuse mutation by MASC

```
java -jar MASC.jar Cipher.properties
```

Listing 4: Running MASC CLI with a configuration file

Benign developer, harmful fix (T2): A benign developer who is trying to address a vulnerability identified by a crypto-detector in good faith, but ends up introducing a new vulnerability instead.
Evasive developer, harmful fix (T3): A developer who aims to finish a task as quickly or with low effort (*e.g.,* a contractor), and is hence attempting to purposefully evade a crypto-detector.

The restrictive operators mutate the restrictive values that abstracts away the crypto-API misuse. For example, the abstraction can be based on method chaining, changing letter case (JCA is case-insensitive), and introducing alias variables, as shown in Listing 1. 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,

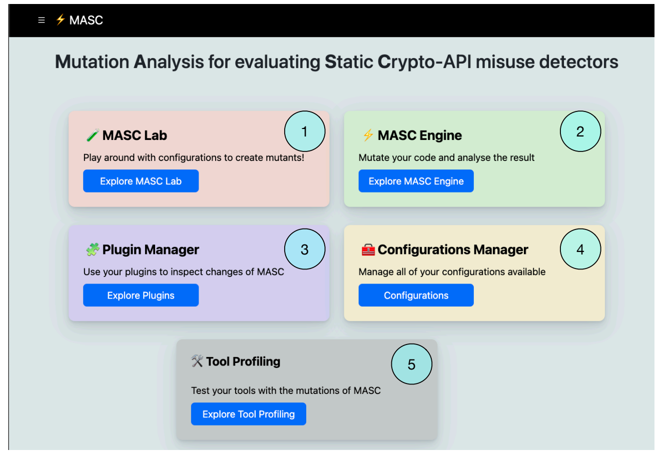

Figure 3: Web based Front-end of the MASC

(2) Mutate open source applications by uploading the zipped source code in MASC Engine,
(3) Use custom implemented mutation operators as plugins,
(4) Create and upload configuration files, and
(5) Profile crypto-detectors by analyzing caught and uncaught mutants.

The detailed description of each of these, with example configuration files, and detailed developer documentation is shared in the open-source repository of MASC [2].

6 FUTURE WORK AND CONCLUSION

We discussed the overview, design goals, implementation details and usage of MASC, a user-friendly tool for mutation-based evaluation of static crypto-API misuse detectors. While we do not report any additional crypto-detector evaluation in this demonstration paper, evaluation results of the original implementation of MASC are available in the original paper [3]. We plan to evaluate additional crypto-detectors with the current implementation of MASC, and aim to extend the customization support to the additional scopes, *i.e.,* exhaustive scope and similarity scope. We hope that the current implementation of MASC will help crypto-detector stakeholders, *i.e.,* security researchers, developers and users, to systematically evaluate crypto-detectors. Furthermore, we envision that that open-source enthusiasts will augment the mutation operators of MASC further, empowered by its easy to extend architecture, thus helping improve crypto-detectors by finding novel flaws.

ACKNOWLEDGMENTS

This work is supported in part by NSF-1815336, NSF-1815186, NSF-1955853 grants and Coastal Virginia Center for Cyber Innovation and the Commonwealth Cyber Initiative, an investment in the advancement of cyber R&D, innovation, and workforce development. For more information about COVA CCI and CCI, visit www.covacci.org and www.cyberinitiative.org.

REFERENCES

[1] Secure Platforms Lab 2022. *MASC Artifact.* Secure Platforms Lab. Retrieved May, 2023 from https://github.com/Secure-Platforms-Lab-W-M/MASC-Artifact

[2] Secure Platforms Lab 2023. *MASC*. Secure Platforms Lab. Retrieved May, 2023 from https://github.com/Secure-Platforms-Lab-W-M/MASC

[3] Amit Seal Ami, Nathan Cooper, Kaushal Kafle, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. 2022. Why Crypto-detectors Fail: A Systematic Evaluation of Cryptographic Misuse Detection Techniques. In *2022 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, San Francisco, CA, USA, 397–414. https://doi.org/10.1109/SP46214.2022.9833582

[4] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[5] CogniCrypt. 2020. *CogniCrypt - Secure Integration of Cryptographic Software | CogniCrypt*. https://www.eclipse.org/cognicrypt/ Accessed June, 2020.

[6] CryptoGuard. 2020. *Oracle - Industrial Experience of Finding Cryptographic Vulnerabilities in Large-scale Codebases*. https://labs.oracle.com/pls/apex/f?p=94065:40150:0::::P40150_PUBLICATION_ID:6629 Accessed July, 2020.

[7] GitHub. 2020. *Announcing third-party code scanning tools: static analysis & developer security training - The GitHub Blog*. https://github.blog/2020-10-05-announcing-third-party-code-scanning-tools-static-analysis-and-developer-security-training/ Accessed Nov, 2020.

[8] "Java". 2020. Java Cryptography Architecture (JCA) Reference Guide. https://docs.oracle.com/en/java/javase/11/security/java-cryptography-architecture-jca-reference-guide.html#GUID-815542FE-CF3D-407A-9673-CAE9840F6231

[9] lgtm. 2020. LGTM - Continuous Security Analysis. https://lgtm.com/ Accessed Nov, 2020.

[10] OASIS. 2021. The Static Analysis Results Interchange Format (SARIF). https://sarifweb.azurewebsites.net/ Accessed Jul, 2021.

[11] owasp. 2020. Test Cases for Risky or Broken Cryptographic Algorithm Erroneously Labeled as Not Vulnerable · Issue #92 · OWASP/Benchmark. https://github.com/OWASP/Benchmark/issues/92 Accessed Nov, 2020.

[12] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. 2019. CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security - CCS '19*. ACM Press, London, United Kingdom, 2455–2472. https://doi.org/10.1145/3319535.3345659

[13] Xanitizer. 2020. Xanitizer by RIGS IT - Because Security Matters. https://www.rigs-it.com/xanitizer/ Accessed May, 2020.

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