A Reevaluation of Why Crypto-detectors Fail: A Systematic Revaluation of Cryptographic Misuse Detection Techniques

Scott Marsden

Sherman Oaks, California, United States of America

Bachelor of Science, Elon University, 2020
Master of Science, The College of William & Mary, 2023

A Thesis presented to the Graduate Faculty
of The College of William & Mary in Candidacy for the Degree of
Master of Science

Department of Computer Science

The College of William & Mary
April 2023

APPROVAL PAGE

This Thesis is submitted in partial fulfillment of
the requirements for the degree of

Master's of Science in Computer Science

Scott Marsden

Approved by the Committee, April 2023

Committee Chair
Professor Denys Poshyvanyk, Computer Science
College of William & Mary

Assistant Professor Dmitry Evtyushkin, Computer Science
College of William & Mary

Assistant Professor Adwait Nadkarni, Computer Science
College of William & Mary

COMPLIANCE PAGE

Research approved by

Protection of Human Subjects Committee

Protocol number(s): PHSC-####-##-##-#####-protocol
PHSC-####-##-##-#####-protocol

Date(s) of approval: mm/dd/yyyy
mm/dd/yyyy

ABSTRACT

The correct use of cryptography is central to ensuring data security in modern software systems. Hence, several academic and commercial static analysis tools have been developed for detecting and mitigating crypto-API misuse. While developers are optimistically adopting these crypto-API misuse detectors (or crypto-detectors) in their software development cycles, this momentum must be accompanied by a rigorous understanding of their effectiveness at finding crypto-API misuse in practice. The original paper presents the MASC framework, which enables a systematic and data-driven evaluation of crypto-detectors using mutation testing. MASC was grounded in a comprehensive view of the problem space by developing a data-driven taxonomy of existing crypto-API misuse, containing 105 misuse cases organized among nine semantic clusters. 12 generalizable usagebased mutation operators were developed and three mutation scopes that can expressively instantiate thousands of compilable variants of the misuse cases for thoroughly evaluating crypto-detectors. Using MASC, nine major crypto-detectors were evaluated and 19 unique, undocumented flaws that severely impact the ability of crypto-detectors to discover misuses in practice were found.

For my thesis I built upon this previous research and greatly expanded the MASC framework. MASC was expanded in all areas by adding new functionality, new operators, new misuses, and expanding the taxonomy. In addition, I reevaluated the most up to date versions of the original 9 crypto-detectors and evaluated 5 additional crypto-dectors. On top of this I also doubled the amount of applications I used to evaluate the tools.

To analze crypto-dectors I looked at both the 9 crypto-detectors evaluated in the original work and 5 new crypto-detectors. For the original crypto-detectors I evaluated them with the updated MASC against their most up to date versions to determine if old flaws that have previously been fixed have a tendency to reappear. Both old and new crypto-detectors were evaluated with mutated Android and Java applications that were used in the original MASC paper, 15 newly mutated Android and Java applications, and minimal examples of cryptographic misuse.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I would like to first acknowledge Amit Ami for helping guide me through the thesis. We met nearly weekly to help ensure I was making progress and that it was consistent with the original work. Amit was a great help as both a mentor and someone who could double check my work.

I would also like to thank the developers of all the tools that were evaluated within the paper. Without these developers working to provide these tools I would not be able to find ways to improve them. I also appreciate that they were open to a discussion to find ways to improve the tools.

Finally, for early portions of the project I worked in collaboration with other students for class projects. These other students are Trevor Stalnaker, Johnny Clapham, and Jake Zappin I would like to thank them as well for contributing to the project. In addition, I would also like to thank Syed Yusuf Ahmed and Radowan Mahmud Redoy for adding additional features to the automated analysis component of MASC.

Insert heartfelt dedication here...

LIST OF TABLES

LIST OF FIGURES

A Reevaluation of Why Crypto-detectors Fail: A Systematic Revaluation of Cryptographic Misuse Detection Techniques

Chapter 1

Introduction

Cryptography is essential in making sure data is confidential and secure in the modern world. Making sure cryptography primitives are used correctly, however, has always been difficult problem to solve whether it be in critical systems like banking or the widespread misuse of cryptographic API found in mobile and web apps. These misuses can lead to vulnerabilities that allow for data leaks. To solve this problem researchers have designed tools that can be integrated into the development pipeline that catch these misuses and allow developers to find the problems before they release software to the public. These tools are known as crypto-API misuse detectors also known as crypto-detectors. These crypto-detectors are vital to helping keep development of applications secure. However, these crypto-detectors are not without their own faults.

crypto-detectors see widespread use across IDEs, corporate testing suites, by source control platforms (such as GitHub), and comercial use. crypto-detectors are relied upon by many developers that deal with sensitive information, there is an expectation and an assumption that they are reliable. Developers utilize these crypto-detectors to help have peace of mind that their code is now more secure. However, a lot is still unknown about how effective they are at crypto-API misuse despite how eager developers are to adopt them. The MASC framework [**?**], presented in the orignal paper tried to provide a

solution to this problem by providing a framework to evaluate crypto-detectors beyond simple manually created benchmarks.

As described in the original paper MASC is "the first systematic, data-driven framework that leverages the well-founded approach of Mutation Analysis for evaluating Static Crypto-API misuse detectors." MASC is used by a user in a similar manner to typical mutation analysis. MASC is capable of "mutating Android/Java apps by seeding them with mutants" containing crypto-API misuse. The mutated apps can then be analyzed by a crypto-detector and seeing which mutants are not located during the analysis reveals both design and implementation flaws in that crypto-detector.

For my thesis I took the already existing MASC framework and extended it. This was done by adding a variety of new functionality and greatly expanding the functionality that already existed. I expanded MASC in both depth and width. The updated MASC framework was also used to reevaluate crypto-detectors that were evaluated by the original paper and evaluate five additional crypto-detectors. With the new additions to MASC and new evaluation I was able to determine: do misuses that were reported and fixed have a tendency to reappear in future builds, have crypto-detectors improved since the original paper, can MASC discover new flaws in crypto-detectors, what are the characteristics of these flaws, and what is the impact of the flaws on the effectiveness of crypto-detectors in practice?

1.1 Overview

The original MASC framework was built based on a Crypto-API Misuse Taxonomy. At the time is was the first comprehensive taxonomy of crypto-API misuse cases containing 105 cases. These were found using a data driven process that systematically identified, studied, and extracted misuse cases from both academic and industrial sources published from 1998-2018. This taxonomy provided the main building block for designing mutants that can emulate realistic misuses. For my thesis we expanded this taxonomy to cover

papers from 2019-2022. The extended taxonomy includes 19 more papers and 13 more misuses.

MASC also had several Crypto-Mutation Operators and Scopes. These different abstractions were designed to allow flexibility in MASC to create a large variety of feasible mutants. The threat model MASC is based on consists of three types of threats that crypto-detectors are likely to face: benign developer accidental misuse (T1), benign developer harmful fix (T2), and evasive developer harmful fix (T3). In addition, MASC also consisted of usage base mutation operators which were general operators that contain common characteristics that can leverage a diverse set of crypto APIs to create misuse cases that can be found within the taxonomy. These operators consist of two main categories: restrictive, where a developer can only instantiate certain objects by providing values from a predefined set such `Cipher.getInstance(<parameter>)` and flexible which consists of operators with a large amount of extensibility such as developers can customize the hostname verification component of the SSL/TLS handshake by creating a class extending the HostnameVerifier, and overriding its verify method, with any content. The original set of operators consisted of 12 operators (six restrictive and six flexible). For my thesis I focused on expanding the amount of restrictive operators since more are necessary to fully represent the taxonomy. This is due to the fact that restrictive operators take in a specific set of inputs from the user and represent one or more misuse cases while the flexible operators are designed to represent many. Due to this focus was placed on the expansion of restrictive operators since more are necessary to capture the whole taxonomy. I added seven additional restrictive operators to the total count based both on misuses that were not represented from the original taxonomy and to represent the extended taxonomy. I also added restrictive operators to further extend representation of the evasive developer (T3) since the original work was intentionally conservative with imagining what evasive developers were capable of. After more research it was determined that new operators could be created to emulate more evasive behavior while still being statically computable.

MASC also includes three types of mutation scopes that allow for seeding of mutants variable fidelity to realistic API-use and threats.

I also expanded MASC to contain several new features. These consist of adding an automated analysis tool and the sensitivity evaluator. The automated analysis provides MASC with a way to automatically evaluate a crypto-detector's output and determine which misuses were caught or not. It leverages Static Analysis Results Interchange Format (SARIF) files (a file type similar to JSON) to determine where misuses were. It is also designed to catch flaws with crypto-detectors based on output. The new component can test a variety of cases against a crypto-detectors that range in complexity and will report if a simple case fails to be caught and stop running, since a more complex case cannot be feasibly caught if the simple version fails. Since all evaluation from the original paper was done by hand and was very time-consuming to ensure accuracy. This tool was designed to help researchers and users save time while evaluating crypto-detectors and ensure accuracy. This tool was further expanded by other developers to be capable of doing full end to end analysis with crypto-detector by running the crypto-detector as a part of MASC.

The other main new feature introduced as a part of MASC is the sensitivity evaluator. This tool defined the different types of sensitivities commonly associated with crypto-detectors: flow, object, context, path, and alias sensitivity. Certain tools claim to be built with these sensitivities in mind. These were used to categorize all the operators contained under one or more of these categories. MASC can then be run and will generate mutants only for the specified sensitivity type. This allows for a lower barrier to entry for users and to allow users to test mutants specifically against the claims of a crypto-detector.

For the evaluation of crypto-detectors I used the same methodology as the original paper but extended the number of operators and the number of applications. I evaluated eight of the major crypto-detectors used in the original paper and also evaluated five addition crypto-detectors. The original paper also evaluated the crypto-detectors with a 15 different Android/Java applications and a group of minimal cases. All of these same applications and minimal cases were used again in evaluation with the addition of 16 new

Android/Java applications that were mutated using the newly extended MASC as well as a group of new minimal cases to test the new operators. Additionally, my findings were disclosed to the designers/maintainers of the tools.

1.1.1 Motivation and Background

The motivation for extending MASC remains consistent with the original motivation behind MASC, "Insecure use of cryptographic APIs is the second most common cause of software vulnerabilities after data leaks." Many developers are likely to use crypto-detectors in their development pipeline since they are not experts in the area but wish to catch vulnerabilities before releasing software. This means that the reliability of crypto-detectors and their ability to locate misuses directly impacts the security of the end user and their data. Evaluating crypto-detectors is an ongoing problem. New misuses are discovered frequently and it is important to have a tool that can keep up with new misuses. The idea behind expanding the MASC framework came from the need to keep up with new misuses and still be a reliable tool for evaluating crypto-detectors.

The crypto-detectors evaluated in the original paper needed to be reevaluated because bugs that get fixed have a tendency to reappear in future patches.[**?**] The newest versions of these tools needed to be reevaluated to see if they have made any improvements with catching misuses and ensure that flaws that were found have truly been fixed. The set of evaluated crypto-detectors was also expanded so that flaws can be reported to other widely used tools to help improve them as well. In addition, since the original paper was published new tools have appeared and gained traction such as Amazon CodeGuru. I wanted to ensure MASC conducted an evaluation on as many crypto-detectors as possible. Since MASC's viability was proven in the original work it was important to extend and improve the work as well as extend its reach. The MASC framework, like many software engineering projects, is an ever evolving framework that is designed to be a reliable way to help evaluate the effectiveness of crypto-detectors.

The motivating example for this new component of the research would be a scenario like the following. Consider Alice, a Java Developer who uses SonarQube as a part of her development pipeline, a known state of the art crypto-detector known for being able to detect security vulnerabilities found in software prior to release. Alice had reported that there was flaw in the crypto-detector that allowed the following line of code to pass without detection:

```
Cipher cipher = Cipher . getInstance ( " des " );
```

She was told that in the new version of the tool that this would be fixed. Her team upgrades to the latest version and sees that it now detects this misuse. Then some time goes by and another new version is released. Alice's team upgrades to this but one of her team members, Bob, puts the same line of code in the software unaware that it is a misuse. It is able to get through to production since the crypto-detector did not detect it even though Alice reported it previously and is under the belief that it is no longer present.

In addition I continued using the motivation from the original paper of Alice being an unaware developer using:

```
Cipher cipher = Cipher . getInstance ( " des " );
```

and it not being detected by the crypto-detector. DES is the common version of the misuse, however, Java supports both the uppercase and lowercase versions of this misuse. A crypto-detector would be expected to detect this common mistake.

1.1.2 Threat Model

The threat model remains the same between both this extension of work as well as the original. In the original paper they defiend the scope of their evaluation by leveraging the documentation of popular crypto-detectors to uderstand how to position their tool and what they claim to be capable of. When evaluating the new tools I looked through their documentation as well and ensured that they made similar claims as the originally evaluated crypto-detectors to ensure the model was consitent across all evaluated crypto-detectors. As an example snyk's documentation claims "Empower developers to become

quasi-security professionals with Snyk Code's comprehensive security tooling." [?] Once again similarly Amazon Code Guru claims "Our security detectors use machine learning and automated reasoning to analyze data flow to perform whole-program inter-procedural analysis, across classes, methods, and files to detect hard-to-find security vulnerabilities." [?] and "Java Crypto Library Best Practices help you check common Java cryptography libraries, such as Javax.Crypto.Cipher, to identify that they are initialized and called correctly" or SonarQube says "maximum protection with taint analysis." [?] All five of the newly evaluated crypto-detectors make similar claims of security assurance. Just like the original nine crypto-detectors that were evaluated and now reevaluated they need to be evaluted by a 3rd party to verify their claims and assure developers that they can truly do what claim they can.

For my threat model I continued off the same model used in the original paper. It assumed that there are some circumstances where they might be deployed in adversarial circumstances due to their claims of being useful for security audits and similar tasks. There are some circumstances where security audits are required and one of the parties involved is opposed to this such as verification for deploying an app in the Google Play Store. To reiterate the threat model consists of three types of adversaries (T1-T3). This threat model is how components of MASC are designed and guide how evaluation was conducted:

T1 Benign developer, accidental misuse - this model assumes the developer accidentally misuses crypto-API, but attempts to detect and address the vulnerabilities with the assistance of crypto-detector.

T2 Benign developer, harmful fix - This scenario shares some similarities to the first as it assumes the developer does not fully understand crypto-API but they are using a tool to help identify misuses. When a misuse gets flagged the developer attempts to fix it but introduces a new vulnerability in its place.

T3 Evasive developer, harmful fix - This assumes the developer is trying to finish the task quickly or with low effort and is intentionally trying to evade the crypto-detector.

When the alert from a detector is received the developer will do quick fix that results in potentially hiding the misuse rather than truly fixing it.

Once again like the original design of MASC the threats are meant to mimic how crypto-detectors have to operate in practice and the evaluation is designed based on what the crypto-detectors should be detecting. However, there is a gap between what should be and what is. This is why just like the design of the original MASC I kept all use cases in consideration.

However, I did try to further capture the T3 developer compared to the original work. In the original work the T3 developer was intially designed to be somewhat conservative to ensure that cases were as close to reality as possible. The original developer were aware that they could go further but they wanted to be more conservative than some real life examples they had seen. For this extension it was discovered that there are cases far more complex and malicious than was originally realized. So in the expansion of MASC some operators were added to include more complex T3 cases.

1.2 Related Works

Security researchers have recently shown significant interest in the external validation of static analysis tools [**?** **?** **?** **?** **?** **?**]. Particularly, there is a growing realization that static analysis security tools are sound in theory, but *soundy in practice*, *i.e.*, consisting of a core set of sound decisions, as well as certain strategic unsound choices made for practical reasons such as performance or precision [**?**].

Sound*y* tools are desirable for security analysis as their sound core ensures sufficient detection of targeted behavior, while also being *practical*, *i.e.*, without incurring too many false alarms.

However, given the lack of oversight and evaluation they have faced so far, *crypto-detectors* may violate this basic assumption behind sound*i*ness and may in fact be *un-sound*, *i.e.*, have fundamental flaws that prevent them from detecting even straightforward

instances of crypto-API misuse observed in apps. This intuition drives our approach for systematically evaluating crypto-detectors, leading to novel contributions that deviate from related work.

To the best of our knowledge, `MASC` is the first framework to use mutation testing, combined with a large-scale data-driven taxonomy of crypto-API misuse, for comprehensively evaluating the detection ability of crypto-detectors to find design/implementation flaws. However, in a more general sense, Bonett et al. [**?**] were the first to leverage the intuition behind mutation testing for evaluating Java/Android security tools, and developed the μSE framework for evaluating data leaks detectors (*e.g.*, FlowDroid [**?**] and Argus [**?**]). `MASC` significantly deviates from μSE in terms of its design focus, in order to address the unique challenges imposed by the problem domain of crypto-misuse detection. Particularly, μSE assumes that for finding flaws, it is sufficient to *manually* define `"a"` security operator and strategically place it at hard-to-reach locations in source code. This assumption does not hold when evaluating crypto-detectors as it is improbable to cast cryptographic misuse as a single mutation, given that cryptographic misuse cases are diverse, and developers may express the same type of misuse in different ways. For example, consider three well-known types of misuse that would require unique mutation operators: (1) using `DES` for encryption (operator inserts prohibited parameter names, *e.g.*, `DES`), (2) trusting all SSL/TLS certificates (operator creates a malformed `TrustManager`), and (3) using a predictable initialization vector (IV) (operator derives predictable values for the IV). In fact, developers may even express the same misuse in different ways, necessitating unique operators to express such distinct *instances*, *e.g.*, the `DES` misuse expressed differently in Listing **??** and Listing **??**. Thus, instead of adopting μSE's single-operator approach, `MASC` designs general usage-based mutation operators that can expressively instantiate misuses from our taxonomy of misuses. In a similar manner, `MASC's` contextualized mutation abstractions (*i.e.*, for evaluating crypto-detectors) distinguish it from other systems that perform vulnerability injection for C programs [**?**], discover API misuse using mutation [**?** **?**], or

evaluate static analysis tools for precision using handcrafted benchmarks or user-defined policies [? ?].

Finally, the goal behind MASC is to assist the designers of crypto-detectors [? ? ? ? ? ? ? ? ? ? ? ?] in identifying design and implementation gaps in their tools, and hence, MASC is complementary to the large body of work in this area. Particularly, prior work provides rule-sets or benchmarks [? ?] consisting of a limited set of cryptographic "bad practices" [?], or taxonomies of smaller subsets (*e.g.*, SSL/TLS misuse taxonomy by Vasan et al. [?]). However, we believe that the new extended taxonomy is the most systematically-driven and comprehensive taxonomy of *crypto-API* misuse, that is further expanded upon into numerous unique misuse instances through MASC's operators. Thus, relative to prior handcrafted benchmarks, MASC can thoroughly test detectors with a far more comprehensive set of crypto-misuse instances.

In addition to the previous work while working on the extension I also looked at other tools trying to accomplish similar goals to MASC such as [?] and [?]. I compared what we have done to what these tools are trying to do to ensure that MASC is still state of the art and is keeping up to date with other tools. MASC is still capable of doing more with mutation testing and its operators than these other tools. I still believe MASC is the state of the art tool for identifying flaws in Crypto-Detectors especially after the extension of work.

1.3 Bibliographical Notes

The paper supporting the content described in this Chapter was written in by other members of the SEMERU group at William & Mary:

Ami, A., Cooper, N., Kafle, K., Moran, K., Poshyvanyk, D., and Nadkarni, A., "Why Crypto-detectors Fail: A Systematic Evaluation of Cryptographic Misuse Detection Techniques", in Proceedings of IEEE Symposium on Security and Privacy (SP'22) May 22, 2022, pp. 614-631

This paper is the basis for the extension described throughout this work. MASC is also moticated partially by another project called µSE done by the SEMERU group at William & Mary:

Ami, A., Kafle, K., Moran, K., Nadkarni, A., and Poshyvanyk, D., "Systematic Mutation-based Evaluation of the Soundness of Security-focused Android Static Analysis Techniques", ACM Transactions on Security & Privacy (TOPS), vol. 24, no. 3, February 2021, pp. 1-37

Bonett, R., Kafle, K., Moran, K., Nadkarni, A., and Poshyvanyk, D., "Discovering Flaws in Security-Focused Static Analysis Tools for Android using Systematic Mutation", in Proceedings of 27th USENIX Security Symposium (USENIX'18), Baltimore, MD, USA, August 15-17, 2018, pp. 1263-1280

Ami, A., Kafle, K., Moran, K., Nadkarni, A., and Poshyvanyk, D., "µSE: Mutation-based Evaluation of Security-focused Static Analysis Tools for Android", in Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE'21), Formal Tool Demonstration, Virtual (originally Madrid, Spain), May 25th - 28th, 2021, pp. 53-56

Chapter 2

The MASC Framework

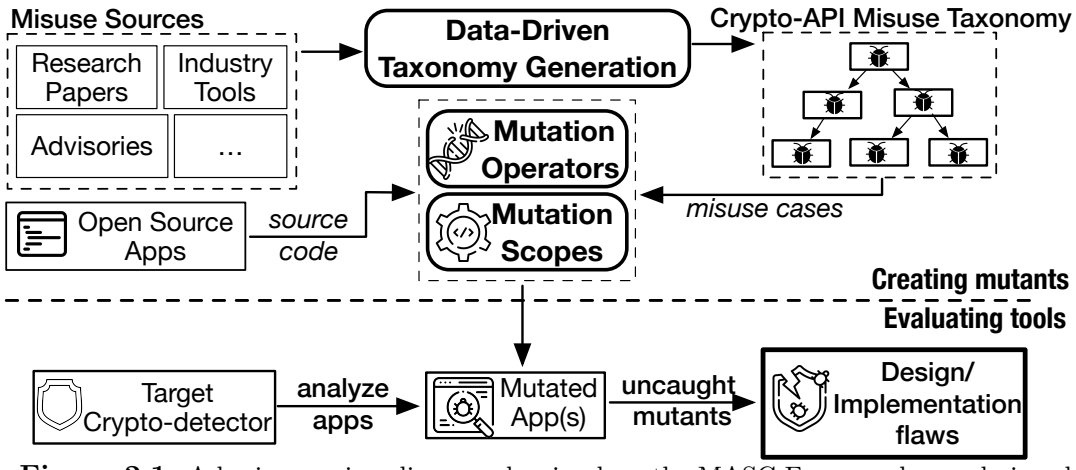


Figure 2.1: A basic overview diagram showing how the MASC Framework was designed

2.1 Overview

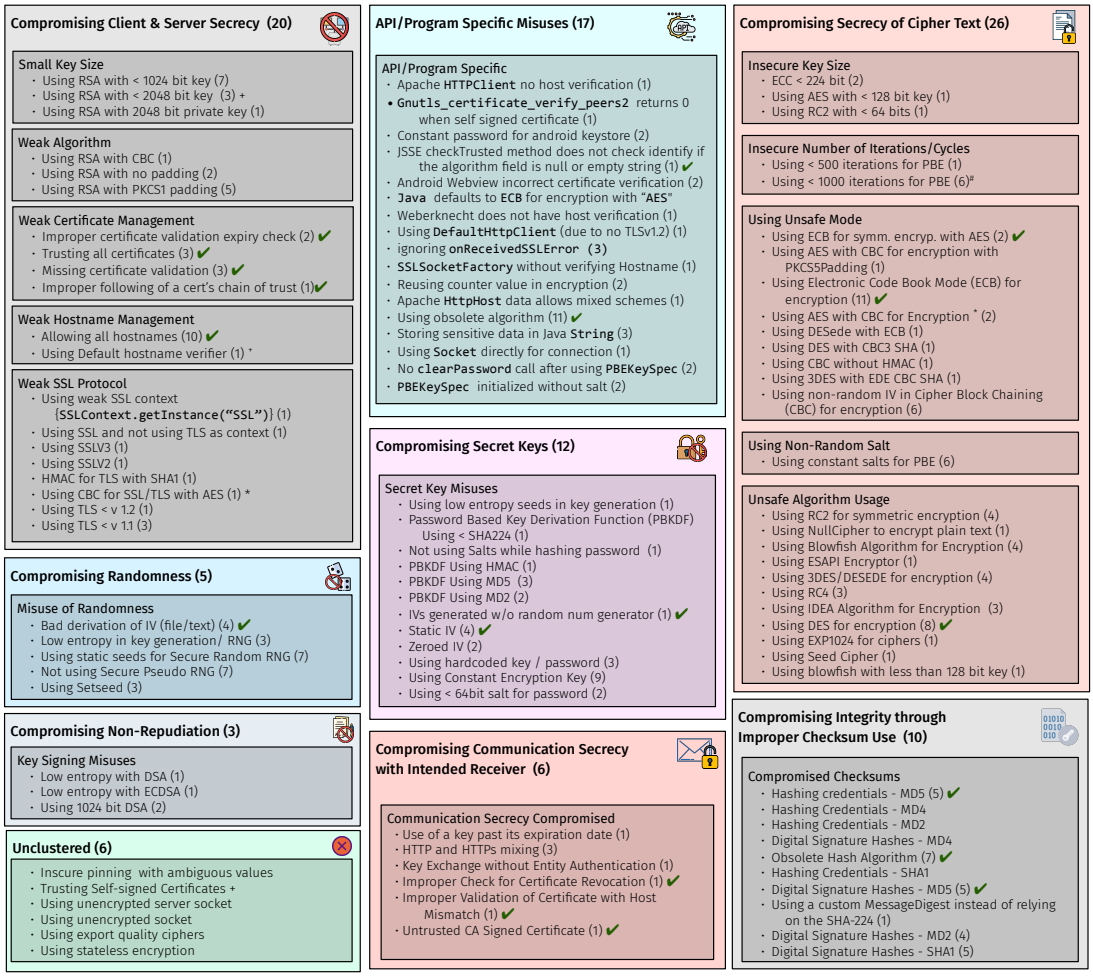
The original work proposed a framework for Mutation-based Analysis of Static Crypto-misuse detection techniques (or MASC). Figure 2.1 above shows the design of the MASC framework as described in the original work. The framework remains the same and all the ideas from this original framework were intentionally consistent throughout my work as well. Cryptographic libraries contain many sets of API with a variety of potential misuse cases, this resulted in an incredibly large design space. Due to this the original

work decided to start MASC by creating a "data-driven taxonomy crypto API misuse." This allows for the misuse cases to be grounded based on what is seen in practice. This both allows for a more focused design space and helps justify the misuses MASC puts into practice. For this extension the taxonomy was expanded to include more cases than the prior work and updated with newer misuses and additional confirmation of previous misuses.

Misuses had to be handled in a way that allowed for some flexibility since misuses can appear in a variety of ways. For example the original paper provides the example of DES can be provided as a variable in Cipher.getInstance(<parameter>) and can also be provided in lowercase. To be able to express both of these without hard coding them as examples MASC is designed by defining usage-characteristics of cryptographic APIs. These can then be leveraged to design "general, usage based mutation operators." These operators being designed in this way allows for a single operator to be able to handle a variety of misuses. This was made clearer when the taxonomy was expanded due to the fact that it was found that some operators previously designed for MASC were already capable of expressing these misuses with little to no effort. A substantial amount of work was put into extending operators to cover misuses that were not already represented in both the original and extended taxonomy.

Instantiating mutants is done by applying the mutation operators to the cases found in the taxonomy, MASC injects the mutants into Java/Android applications. How mutations are seeded is based on one of three scopes: the main scope, exhaustive scope, and similarity scope. These scopes are designed to emulate practical scenarios described in the threat model. Once mutated the apps can then be used to analyze a crypto-detector targeted for evaluation. Based on the results produced by the target it is possible to determine based on undetected mutants where design and implementation flaws exist. This is especially true in the case of reemerging cases.

2.1.1 Extended Taxonomy



* CBC is insecure in TLS/client-server context; + applicable in specific situations; some misuse are newer compared to other in same cluster, # PKCS5 suggestion based

Figure 2.2: The orignal derived taxonomy of cryptographic misuses the extended taxnonmy is based on. (n) indicates misuse was present across n artifacts. A ✓indicates that the specific misuse case was instantiated with `MASC`'s mutation operators for our evaluation.

In expanding MASC as a framework it was necessary to update the taxonomy to bring it up to date from when the original taxonomy was created in 2018. Above in Figure 2.2 the original taxonomy is shown and was the basis of this extension. Since the taxonomy is serving as expansion rather than a replacement the same methodology was used from its initial creation to extend it. This was done to ensure the taxonomy as a whole was consistent and one whole work rather than two different taxonomies. In order to locate academic and research papers related to Crypto-API misuse, it was first necessary to

Table 2.1: New misuse cases added to the taxonomy

Misuse	# Occurrences
Storing sensitive data in Java String	5
Key reuse in stream ciphers	1
Use of expired keys	1
No clearPassword call after using PBEKeySpec	1
Using AES-CTR	1
Weak algorithm for password-based encryption - PBKDF1	1
HMAC for TLS with MD5	1
Using RC5*	1
Using ARCFOUR	1
PBEWithMD5AndDES	1
Hashing Credentials - SHA-224	1
Reusing IVs & key pairs	1
Manually changing hostname verifier	1

identify a set of search terms that would adequately narrow the search space. The authors of the original MASC paper had already identified a sufficiently narrowing set of search terms.

These search terms were used, as well as combinations of these search terms, as a foundation in conducting our searches for academic and industrial papers published between 2019 and 2022 identifying Crypto-API misuse cases. This was also done to ensure the methods used to collect papers were consistent with the methods used in the original paper.

With respect to academic papers, searches were conducted using the search terms laid out in the original work, as well as combinations of those terms on Google Scholar, IEEE Xplore & ACM Digital Library as the primary search engines (these were also the engines used when creating the taxonomy for the original paper). In addition, each citing reference in each of the papers concerning Crypto-API misuse found through searching these databases and search engines was also looked at. By doing this, it was possible to locate approximately 5-6 academic papers, concerning Crypto-API misuse, that were not readily visible through searches alone. This method helped expand the search domain for obtaining relevant academic papers.

For industrial sources, Google, Bing, and StackOverflow were relied upon for searching. While it was possible to locate some relevant documents, they were few and far between compared to the academic papers that we uncovered. In an effort to ensure completeness of the search space, when an industrial document relevant to Crypto-API misuse was located, the references it cited if any were also considered.

Additionally, a search was conducted through the Common Weakness Enumeration (CWE) from the MITRE Corporation [**?**]. The CWE is a database of known hardware and software vulnerabilities, which are classified and categorized. Many crypto-detectors use CWE as a base for misuses that they detect. Notably, it includes known weaknesses that involve software using an API (or Crypto API) in a manner contrary to its intended use. A search was conducted using the CWE for new misuse cases not already identified in the original MASC Taxonomy or not already revealed in academic papers located from 2019 to 2022. Unfortunately, the CWE did not contain any Crypto-API misuses that were not already contained in the MASC Taxonomy or based on our searches of academic papers from 2019 to 2022.

To ensure that selected papers were only relevant to Crypto-API misuse in the MASC Taxonomy extension, the inclusion and exclusion criteria outlined in the original MASC paper was followed. Specifically, to decide whether to consider a paper for further analysis, we used the inclusion criterion that the paper should discuss Crypto-API misuse or its detection. Exclusion criterion was used that the Crypto-API misuse described by the paper will be excluded if it does not relate to the Java programming environment or ecosystem, if the paper was published prior to 2019 or if the paper did not contain relevant information related to the subject of MASC.

In addition to these criteria, an additional exclusion criterion was included specifically with respect to the MASC Taxonomy extension. That is, if a paper discussed Crypto-API-related misuses, but did not identify specific misuse cases in its text, the paper was added to the general list of sources. However, these papers were not included in the final

list of taxonomy sources from which misuse cases were extracted for the MASC Taxonomy extension. A complete list can be seen in Table **??**.

Table 2.2: Papers added to the taxonomy

ID	Title	Year	Venue
37	Ensuring correct cryptographic algorithm and provider usage at compile time [**?**]	2021	ACM
38	Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications [**?**]	2021	USENIX
39	Towards HTTPS Everywhere on Android: We Are Not There Yet [**?**]	2020	USENIX
40	CRYPTOAPI-BENCH: A Comprehensive Benchmark on Java Cryptographic API Misuses [**?**]	2019	IEEE
41	CRYPTOREX: Large-scale Analysis of Cryptographic Misuse in IoT Devices [**?**]	2019	USENIX
42	Negative Results on Mining Crypto-API Usage Rules in Android Apps [**?**]	2019	IEEE
43	Java Cryptography Uses in the Wild [**?**]	2020	arXiv
44	Python Crypto Misuses in the Wild [**?**]	2021	IEEE
45	Understanding How to Use Static Analysis Tools for Detecting Cryptography Misuse in Software [**?**]	2019	IEEE
46	A Comparative Study of Misapplied Crypto in Android and iOS Applications [**?**]	2019	SECRYPT
47	A Dataset of Parametric Cryptographic Misuses [**?**]	2019	IEEE
48	CogniCryptGEN: generating code for the secure usage of crypto APIs [**?**]	2020	IEEE
49	CryptoTutor: Teaching Secure Coding Practices through Misuse Pattern Detection [**?**]	2020	ACM
50	Using Graph Embeddings and Machine Learning to Detect Cryptography Misuse in Source Code [**?**]	2020	ICMLA
51	Evaluation of Static Vulnerability Detection Tools with Java Cryptographic API Benchmarks [**?**]	2022	IEEE
52	Automatic Detection of Java Cryptographic API Misuses: Are We There Yet? [**?**]	2023	IEEE
53	Hotfixing Misuses of Crypto APIs in Java Programs [**?**]	2021	ACM
54	CRYScanner: Finding cryptographic libraries misuse [**?**]	2022	IEEE

After compiling the master list of misuse sources, a misuse case extraction was performed. Two researchers independently identify and record misuse cases present within each of the sources. After this extraction process occurred, the two people met and had what was termed an "agreement/disagreement meeting", as described in the original paper, in order to discuss their findings with respect to our extractions. This was done to ensure consistency across the different papers and ensure that each misuse case was found as well as confirmed. If the two members had any sort of disagreement over if something was a misuse case or not a third independent party was brought in to ensure correctness.

Misuses were extracted and organized with the same methodology as the original paper. The same clusters were used that were designed for the original paper to organize each of the misuses into categories. The clusters were created based on two differentiating criteria: "(1) the security goal/property represented by the misuse case (e.g., secrecy, integrity, non-repudiation) and (2) its level of abstraction within the communication/computing stack (e.g., confidentiality in general, or confidentiality with respect to SSL/TLS)"

In addition to this another misuse extraction was performed at a later date based on the misuses that were extracted. I went through each paper in the taxonomy and located each misuse that was listed for each of these new papers. This was done to confirm the work

that had already been done but ensure that it was thorough. This step was meant to be redundant and ensure the correctness of the taxonomy but resulted in some disagreements of misuses within three of the sources and a disagreement with one of the new sources in the taxonomy as a whole.

The approach for this additional step was for me to look at each individual paper. Based on the updated taxonomy that was produced I would go through the paper and look for each misuse that was specified to be contained within this paper. If the misuse was found it would be marked down and confirmed. If it was not found that misuse in the taxonomy would be flagged and would be brought up again with the original third party. Then if confirmed it would stay in the taxonomy if not it would be removed. Since the taxonomy is an extension I wanted to give extra cation to ensure not only correctness but consistency to ensure the taxonomy was one whole versus being two separate parts.

By the end of this process 13 new misuse cases were identified (see Table **??**) and added to the MASC Taxonomy. In addition, I was also able to further confirm many of the misuses that were already present in the taxonomy with the additional papers.

I believe that this is a solid basis for MASC as a whole and helped give ideas for expansion of operators to ensure that MASC is keeping up with the changing requirements. This taxonomy will likely have to be updated again in the future since security misuses come up frequently and the field is ever-changing. I feel that this taxonomy is a good representation of cryptographic misuses through 2022.

2.1.2 Extended Operators

When designing mutation operators I continued building them based on the trade-off of representing as many misuses cases as possible while also creating a number of operators that can still feasibly maintained. As the project grows this will continue to be a challenge. This is why it is important when operators were designed to make them as flexible to as many misuse cases as possible. This requires a lot of time and planning to ensure the design of the operators is maintainable. The other goal of operators as laid out in the

original work is that they are not designed to exploit general soundiness limitations such as dynamic and implicit calls. This is important to ensure that the results that are found are actual flaws versus something that a tool cannot be reasonably expected to compute. The operators were designed to be expressive of multiple misuse cases to allow for more coverage. Design of operators is also guided by the threat model described previously. When an operator is designed which threat model it covers is considered as well. I felt there was a lot of room to expand upon the evasive developer (T3) so most of the new operators focus on this threat model.

The MASC framework consists of two main types of operators: flexible and restrictive operators. These two types are based on the common characteristics that were found amongst misuses of crypto-API. The restrictive operators are operators where a developer can only instantiate certain objects by providing values from a predefined set such the method Cipher.getInstance(<parameter>) only accepts predefined configuration values for the parameter in String form. While other crypto-APIs allow significant amount of customizability and extendability resulting in the flexible type of operators. For this work I exclusively focused on extending the restrictive operators.

Due to the nature of restrictive operators having more limitations it leaves plenty of room for expansion. In addition, since the taxonomy was expanded it was possible to utilize some of the ideas that already existed in the taxonomy as well as newly discovered misuse case. Since these operators have limitations, there are many possibilities for expanding the restrictive operators. For the extension of MASC we have focused heavily on expanding the number of restrictive operators as well as the functionality of some of the restrictive operators that already existed.

While expanding the operators there was also more of a focus placed on the (T3) cases. The initial paper was intentionally conservative with what was considered an evasive developer. This was due to the work being new and wanting to ensure that the operators were considered fair based on research conducted. After some time and further research I felt that it was possible to push further with this type of developer and after seeing further

examples of evasive developers in real life cases. I wanted to ensure MASC is well-rounded when simulating different types of developers and provide the most use cases possible for crypto detectors.

OP_{13}: *Iterative Method Chaining* – Similar to in MASC OP_5 (Method Chains) this operator implements method chaining to hide the value. Where this operator differs from OP_5 is that it can take a value specified by the user and create that many method calls. The method calls are then chained in succession. Every method call will have a safe value until the final call which transforms the value into an unsafe value. This was created to test the limits of how far crypto detectors check method calls and can allow a user to determine where their fault point is. Just like OP_5 this behavior would simulate a (T3) developer. An example can be seen in Appendix A at **??**.

OP_{14}: *Iterative Nested Conditionals* – This operator shares some similarities with the idea behind OP_{13}: ***b*** – ut instead of using method calls based on the iteration value it creates nested conditionals based on the value. Within the if statement there is an unsafe value and in the else a safe value. The nested conditionals will always pass the unsafe value but like OP_{13}: ***t*** – his operator tests how many nested conditionals a crypto detector can evaluate. An example can be seen in Appendix A at **??**.

OP_{15}: *Method Builder* – This case takes method calls to build the String of a vulnerable call. The object class has methods that each contain a letter such as “D”, “E”, and “S”. Then it has a method that will add the letters together by calling each method and setting them equal to the variable in the class. This variable would now be "DES" and could be passed into an unsafe method. This operator also simulates (T3) behavior and is not something a developer would accidentally do. An example can be seen in Appendix A at **??**.

OP_{16}: *Object Sensitive* – The operator creates two versions of a base object that has method calls to make a String either secure or unsecure. One object sets the variable to secure while the other sets itself to insecure. Then we set the object that is currently secure equal to the object that contains the insecure String. Then the originally secure object

String is passed into the vulnerable API. This is done to test how well crypto detectors can handle object sensitivity. This was designed to give MASC more options for testing object sensitivity since this was an area that was originally lacking. The behavior also fits under our T3 developer. An example can be seen in Appendix A at **??**.

OP_{17}: *Build Variable* – For this operator I converted an insecure String into a char array. Then when the vulnerability is passed into the API the char array is then converted back to a String during the method call. This would fall under the (T1, T2) developer. This is because a developer could be doing some form of String manipulation that requires converting a String to a char array and then passing that into the method call. An example can be seen in Appendix A at **??**.

OP_{18}: *Substring* – The idea behind this case is a user pulling the misuse out of a substring. In this case we took something like "HelloWorldDES" and passed the substring "DES" into the vulnerable API. Once again this is a (T1, T2) developer since they might pull something out of String and pass it in not knowing that it is vulnerable. Since it is not being passed in simply as a String if the crypto detector does not process the change this is easily a misuse that could slip by. An example can be seen in Appendix A at **??**.

OP_{19}: *Static Keystore* – This operator is designed to handle static bytes being passed in insecure ways. For example, if a developer attempted to pass static bytes that are stored in a variable into Android Keystore since this is considered unsecure behavior. This would emulate a T1 developer since this is a very easy mistake to make if someone is unaware of the rules associated with crypto API and vulnerabilities. An example can be seen in Appendix A at **??**.

2.1.3 Threat Based Mutation Scopes

The mutation scopes are designed to emulate the three types of developers described in the threat model. The scopes try to closely simulate placements of vulnerable code for the benign (T1, T2) and evasive developers:

The main scope creates base case mutations and generates simple Java files to be tested. This is used mainly to determine if a crypto-detector is even capable of detecting misuses in the most plain cases. These mutants are seeded in simple Java or Android templates at the beginning of the main method. This ensures that the mutants are found and analyzed. This is meant to simulate the behavior of a benign (T1 and T2) developer.

The exhaustive scope looks at seeding mutations in every possible location such as class definitions, conditional segments, method bodies and anonymous inner class object declarations. Note that it is seeded in places that ensure the app is still compilable. Typically, this is used with a single misuse that is known to be detected by the operator and is used to determine how thorough the crypto-detectors when analyzing files. This scope is meant to analyze a T3 developer they may hide misuses in places that one would not normally expect misuses to appear.

The similarity scope seeds mutants in places where a similar security API is already in use. It emulates taking possibly secure uses and making them insecure while not overwriting the previous code. This emulates where the benign developers (T1 and T2) would potentially place misuses and is meant to test how well the crypto-detectors analyze realistic areas misuses might appear.

Chapter 3

Implementation

The implementation of MASC remains consist with the original work. It involves the same three original components: "(1) selecting misuse cases from the taxonomy for mutation, (2) implementing mutation operators that instantiate the misuse cases, and (3) seeding/inserting the instantiated mutants in Java/Android source code at targeted locations." The extension of the framework was built upon the foundation that was previously laid. All additions remained consistent with the implementation of the original work. A high level diagram of MASC's design is shown in Figure **??**.

1. Selecting misuse cases from the Taxonomy: In the original work 19 misuses were chosen from the taxonomy for mutation utilizing the original 12 operators. Misuses were chosen based on two main factors to ensure that different categories of cryptographic misuse were represented as well as ensuring the more prevalent cases appeared as well. For the extended work only a few additional misuse cases were added such as a static key in android keystore. However, with the expansion of the operators it is possible to create many more mutants with the misuses that were already present. For this extension the previous misuses were leveraged when creating mutations. However, they were designed with other possible misuses in mind similarly to the original 12 operators. 2. Implementing mutants: The mutation operators described in the previous chapter along with the original 12 operators were designed to be applied to one or more crypto-API, for instantiating

specific misuse cases. The goal of generating mutants in programs is to ensure that they are still compilable. To ensure that this remains possible, MASC considered the necessary syntactic requirements of each API that is being implemented (such as the requirement of surrounding try-catch block with appropriate exception handling) and the semantic requirement of the specific misuse that is being instantiated. MASC used Java Reflection to determine all the necessary syntax to automatically create compilable mutants. After this MASC is designed to combine this component with the parameters to create the mutants.

MASC also ensures that mutants generated for evaluation are compilable using two mains steps: "(1) Using Eclipse JST's AST-based to check for identifying syntactic anomalies in the generated mutated apps, and (2) compile the mutated app automatically using build/test scripts provided with the original app." Since the portion is fully automated this is how MASC has made it possible to generate many mutants to evaluate a crypto-detector with little effort from the user.

3. Identifying Target Locations and Seeding Mutants: To locate target locations to seed mutants using the similarity scope the MDroid+ mutation analysis project was leveraged as a component of MASC. For the original work the process it used to determine mutant locations was changed to fit the scope of MASC and support was added to include dependencies that crypto mutations introduce. When I began work on MASC this component was connected to MASC where it needed to be but contained a lot of unnecessary components leftover from the original MDroid+ project that MASC was not utilizing. For the extension I identified the parts of MDroid+ that MASC was using and fully integrated them within MASC. This was done to help consolidate MASC into one project rather than a project using components from two additional projects. The components that were leveraged from MDroid+ still exist but now are fully integrated within MASC.

Similarly, MASC also extended µSE to create the exhaustive scope. The extended µSE was used to find locations where crypto-APIs can be inserted in a program and still

allow the program to be compilable. Just like MDroid+ I integrated the parts of µSE that MASC utilized into the main program.

In addition, the same flaw of corner cases with mutants causing compilation errors still exists within MASC. Due to how MASC is implemented this is something that is not easy to change. These cases still have a change to appear in 0.098% of cases.

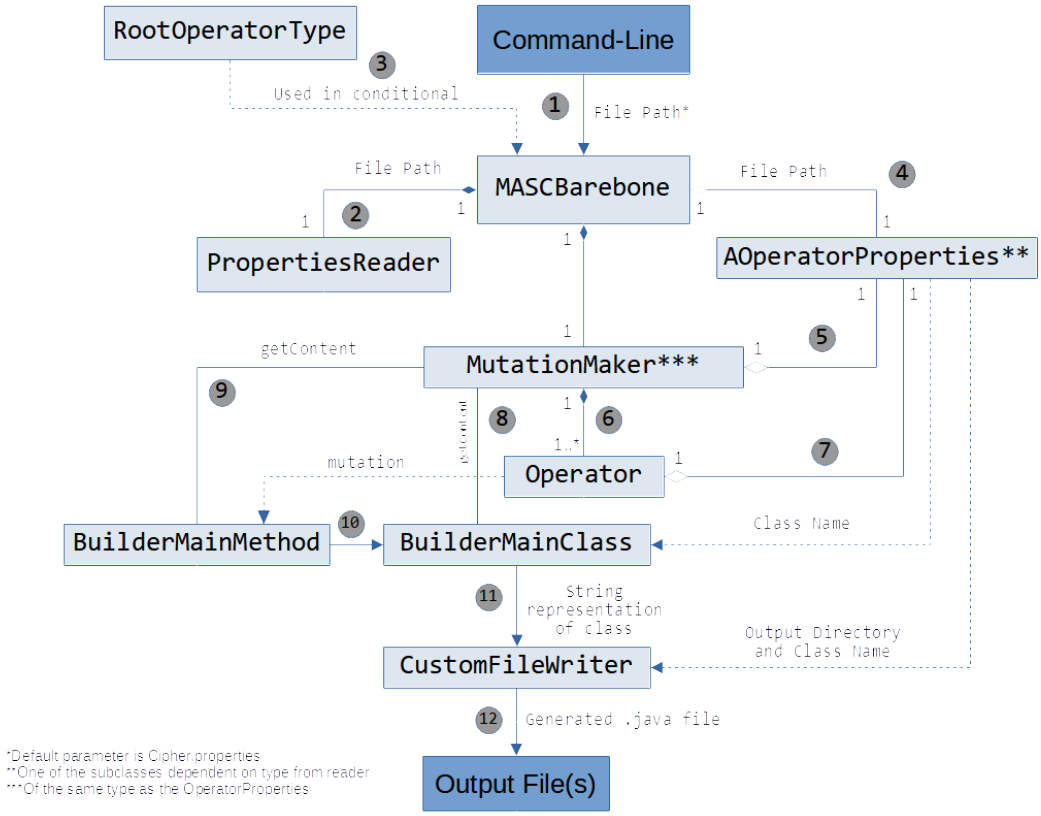


Figure 3.1: A high level diagram showing the architecture design of MASC. This further demonstrates how each component of MASC is designed and how to run the framework

3.1 New Features

While many of the new additions to MASC came in the form of extending work that was already there and heavily extended the evaluation. I also introduced a couple of new features into MASC to help make the framework more user-friendly and allow a new perspective of evaluation to be conducted.

3.2 Sensitivity Evaluator

The sensitivity evaluator is a new component of the MASC framework designed as an alternative way for users evaluate crypto-detectors. This tool is designed to help users access the operators without having to understand the specifics of what each operator does. In security analysis there are sensitivities that are commonly discussed in relation to static analysis tools. These crypto-detectors are built with these sensitivities in mind and claim to be able to logically handle some of them. The main sensitivities I found when looking through past work were flow sensitivity, alias sensitivity, context sensitivity, path sensitivity, and object sensitivity. For this work the sensitivities are defined as the following:

Flow Sensitivity - Flow sensitivity is an incredibly precise form of sensitivity. It recognizes the order that statements are performed and can keep track of the state of the program at that point in time. A flow sensitive analysis performs its analysis based on the sequence of statements. It can tell if two variables are assigned after line 23 while a flow insensitive analysis will only know that the two variables were assigned at some point within the scope of their analysis. A flow analysis will only take into consideration portions of the program that would be run based on the previous lines. Flow sensitivity analysis is an extremely expensive computationally.

Context Sensitivity - Context sensitivity takes into account the information throughout the program when method calls are made to determine if there is a vulnerability. It can differentiate between two different function calls to the same method with different variables. A context insensitive approach would flag both function calls if one of them was considered vulnerable while a context-sensitive approach can differentiate between the two calls. A less sophisticated version of this is interprocedural sensitivity.

Alias Sensitivity - Alias sensitivity is typically a variation of context or flow sensitivity. In Java, alias sensitivity is typically type based, this is because Java is a type safe language. Alias sensitivity is the ability to keep track of a variable that has been aliased to another

variable and still keeping track of the value. If there is a variable named x that equals one and we pass this into a method this passed value would be an alias of x.

Path/Conditional Sensitivity - Path sensitivity only takes into consideration paths through the program that are feasible. It has a heavy focus on things such as conditionals. Within programs some paths or statements can not be reached by the code, a path sensitive analysis would not flag a vulnerability that is unreachable. Path sensitive analysis is only concerned with the path of the program that is possible to be executed.

Field/Index Sensitivity - The ability to differentiate different fields that are a part of the same object. If an object contains two variables one tainted and one that is not tainted and the non-tainted one is called a field sensitive analysis would not flag the object as a vulnerability. This requires keeping track of all the contents of objects separately and understanding when certain aspects of an object are called by the program.

Object Sensitivity - Takes into account different versions of the same object. It has the ability to understand the difference between a version of an object that contains a vulnerability and one that does not contain a vulnerability. If we create two versions of object FOO called f1 and f2 and place a vulnerability in f2 but only interact with f1, an object sensitive tool would be able to recognize if a vulnerability is not present.

With these definitions in mind for the sensitivities I designed the sensitivity evaluator. This tool allows MASC to be run and generate mutants that fit into each specified definition. Once these definitions were clearly defined, I categorized each operator into the category or categories that it fit under. Then with this knowledge I designed the sensitivity runner so that MASC would only produce output based on a specified sensitivity. This was done to lower the barrier of entry for MASC as well as create a new way of evaluating crypto-detectors using MASC. By defining operators in this way it is possible to put more emphasis on cases that fit the description of a crypto-detector. If a crypto-detector made a claim that it was flow sensitive now it is possible to easily run all the cases that are defined to be flow sensitive and see how well it performs against those mutants instead of trying to determine which operators might fit this definition. It would be expected that a

crypto-detector that claims to handle a certain sensitivity would be better suited to handle those cases.

The sensitivity evaluator combines the input of all the various cases and allows the user to specify how they want to run it in one file. It then handles creating all the operators that are related to the selected sensitivity. It will create the operators with the parameters provided by the user within each operator. This tool should make MASC more accessible to those with some knowledge of security without having to get a full grasp about how the operators and specifics of MASC work. I believe both the user interactivity aspect of this and the new perspective will be greatly beneficial to performing research on crypto-detectors. Currently, it only creates minimal examples but was designed to easily be expanded to the other scopes.

3.3 Automated Evaluation

In the original paper all analysis from all the crypto detectors was done by hand, so it required many man-hours and double-checking to ensure there were no errors. To help determine results for researchers and users an additional tool was created for MASC. This tool is the automated analysis. This allows researchers to run MASC on certain crypto detectors and MASC will automatically parse the results and let the user know if the crypto detector failed and how it failed. The tool can also take output from various crypto detectors that were given MASC code and can tell where the crypto detector failed. This is done using a SARIF parser that was created. SARIF stands for Static Analysis Results Interchange Format. This file format is becoming the standard output for Static Analysis tools and is being pushed heavily by GitHub. At the time of creating this tool this file format was fairly new and did not have many tools created to parse its contents. To build the automated analysis component it required first to build a tool that could parse the SARIF format. SARIF format shares a lot of similarities with JSON, this made it possible to leverage some Java JSON libraries such as the Google JSON simple library. I built a

SARIF layer on top of this library to create a tool for parsing output. The SARIF parser tool I created takes in two SARIF files as input one of the code before mutation that was passed into the crypto detector and one SARIF file that was mutated and seeded with misuses. This is done so that if there were any misuses present in the program before MASC was run that are not taken into account and added to the total of misuses the crypto detector found because of MASC.

More specifically, I created a tool that is able to parse a SARIF output from a Crypto-API Misuse Detector to determine which seeded misuse cases were caught by the detector. Put another way, this component checks to make sure the Crypto-API Misuse Detectors were actually able to catch the misuses seeded in the program. Currently, this tool only works for the Main scope of the MASC Framework. However, it can easily be expanded to work with the Exhaustive and Selective scopes in the next extensions of the MASC Framework.

In terms of how the tool works, it takes the SARIF output files obtained from the Crypto-API Misuse Detectors as input along with the MASC configuration file used for mutation. Using this information the tool parses through the MASC configuration file to determine where the created mutated Java files were placed and what the Crypto-API name is being tested. Using this information, the mutated Java files that were created are scanned to find the line that contains the misuse. Once the misuses are found it is then possible to scan through the results of the SARIF file using Flow Analysis.

Flow Analysis in terms of how it is used in this project is the idea that each misuse created by an operator can be represented as a different level of complexity. MASC inherently has levels of complexity already designed into it. To put it simply, a Crypto-API Misuse Detector will have an easier time catching a misuse like "AES" than "A∼ ES".replace("∼","") since the former example is less syntactically complex than the latter example. By designing it this way, the SARIF files can then be analyzed in order of complexity starting with the most basic misuse (or "base case" misuse) and increasing to the more complex ones until the Crypto-API Misuse Detector fails to catch one. If the crypto-detector fails then

the analysis can stop because if a tool cannot find a less complex misuse it would not be expected to find a more complex or mutated misuse case. Implementing the evaluator in this way makes it a lot easier to determine where Crypto-API Misuse Detectors fail and saves a lot of manual effort since this can all be done in an automated fashion. This process also reduces the risk created by manual evaluation.

The tool was later expanded and used a part of the main MASC configuration file. It was made possible that when MASC was run it could run the output it creates against a crypto detector then check that output and let the user know which mutants were found. This additional step was built on top of the SARIF Parsing tool to create the full end to end automated analysis. This step was the integration of the SARIF Parsing analysis tool with the entire main project since the original tool looked at outputs of crypto detectors that had MASC mutants run on them. The new full automated analysis takes that work and brings together with MASC being run as well combines crypto-detectors such as CogniCrypt by running it as a part of the analysis.

Both the SARIF parsing tool and the combination with automated analysis should help users as well as researchers save a lot of time when examining crypto-detectors. This automated analysis is less error-prone and can help automatically determine if a mutation was caught but also what the likely cause of the failure was. It can determine with its Flow Analysis what level of complexity caused the failure and help more easily identify the potential flaws found within detectors. Overall the tool helps MASC become a more complete project and continues to make it more accessible. This helps to further MASC's goal of improving crypto-detectors by better identifying exactly where they fail.

3.4 MASC Web

In another attempt to make MASC user-friendly. I designed the initial version of a website for MASC. Currently, a second version of MASC Web is in active development using Django instead of Flask (which was originally used). The goal of the website was to introduce users

to MASC and allow them to mutate files directly on the website. I integrated MASC into the website and made it so users could upload a file and specify the parameters they wanted to mutate as well as the operator they wanted to use. The website would then provide them with both the original version and the mutated version of the file. Eventually, the goal is to deploy the website to help increase visibility of MASC and allow for additional options for users to access MASC.

Chapter 4

Evaluation and Methodology

The goal of the evaluation was to find flaws in crypto-detectors using MASC once again. The main objectives were to (1) measure the effectiveness of the new operators at uncovering flaws in crypto-detectors and the old ones at uncovering flaws in new crypto-detectors (since the original operators were proven to be effective), (2) continue to learn the characteristics of the flaws found and their real world impacts, and (3) determine how likely flaws are to reappear in future versions of crypto-detectors. Based on these goals to evaluate MASC the same three original research questions were used to conduct evaluation with the addition of another:

RQ1: Can MASC discover new flaws in crypto-dectors?

RQ2: What are the characteristics of these flaws?

RQ3: What is the impact of the flaws on the effectiveness of crypto-detectors in pracitce?

RQ4: How likely is it for flaws to reappear in newer versions of crypto-detectors?

To answer **RQ1-RQ4**, I used the same methodology described in the initial MASC paper. I took the original set of nine crypto-detectors and found their most up-to-date versions if possible (Xanitizer is no longer freely available) and then added an additional five crypto-detectors to the set that were not evaluated in the original paper. The crypto-

detectors that were evaluated were CogniCrypt, SpotBugs with FindSecBugs, QARK, LGTM, GitHub Code Security, LGTM, ShiftLeft Scan, Amazon Code Guru, SonarQube, Codiga, Deepsource, and Snyk. All tools used are under active maintenance since the goal of MASC is to provide feedback on potential improvements to the crypto-detectors it is important that they are still in active use. Similar to the original paper the results of the evaluation are not intended to demonstrate comparative advantages of tools and are not intended as an endorsement of any such tools, each tool is evaluated separately from the others using the same techniques.

For evaluation of the crypto-detectors the steps used to conduct evaluation remained the same. I expanded on the techniques used but ensured they were still consistent with the original work since **RQ1-RQ3** are the same as the original paper.

Step 1 - Selecting and mutating apps: In the original work 13 open source Android apps and four sub-systems of Apache Qpid Broker-J were found and used for mutation. For the extension these same apps with their original mutations were used as a part of the evaluation. In addition, I located 15 open source Android apps on Github and included Tink, a Java project from Google, for mutation. To locate these additional candidates for mutation I used a similar methodology as the original work. The way the new Android apps were located was by using GitHub's advanced search to locate Android apps that had at least 200 stars, these were then sorted by how recently they were updated to help ensure they would be both compilable and up to date with dependencies. All apps that were used were tested to ensure they were compilable prior to mutation. In addition to help get a variety of different types of apps I also searched for apps that contained tags such as: security, cryptography, secure, games, tools, calendar, exercise, and location. Out of the 15 new Android apps, five of them were specifically found to include crypto-graphic API's (Cipher, MessageDigest, X509TrustManager) and were mutated using the Similarity scope. In addition to the 20,303 mutations found in the original MASC evaluation, I introduced 26,765 mutants within the new Android apps and Java project. This gives a grand total of 47,068 mutants generated by MASC used for evaluation, more than double the original

amount of mutants. Generating all the new mutants took about 20 minutes total on MASC which addresses RQ3, and did not require any human intervention. MASC is capable of generating mutants very quickly the bottleneck of time comes from crypto-detectors evaluating the mutants.

Step 2 - Evaluating crypto-detectors and identifying unkilled/undetected mutants: To conduct the evaluation on a crypto-detector I analyzed the mutants that were produced using the crypto-detector. Afterwards, mutants that were not detected by the crypto-detector were identified and flagged as unkilled. To conduct this full evaluation I kept track of logs produced by MASC for all the apps, this log contained information such as line numbers where mutations were placed and what type of mutations were placed in those locations. Initially I ran the unmutated version of the app against each crypto-detector. Using the output produced I made a note of any cryptographic related misuses that were found on the unmutated version to ensure that these were not counted as a mutation produced by MASC. Then the mutated version of the app could be analyzed by the crypto-detector and results could be compared. Once the similarities of the reports were eliminated I looked at the remaining results and any remaining cryptographic misuse that was found is considered a mutant inserted by MASC that was killed by the crypto-detector. Each killed mutant is confirmed by comparing it with the log to ensure that a mutant was inserted at this location and the correct flag was produced by the crypto-detector. In addition, this analysis was also conducted automatically by the automated analysis mentioned in section 3.0.3 for any tool that could produce SARIF output such as CogniCrypt. The approach taken for this is done to ensure that misuses found by the crypto-detector that were not inserted by MASC are not considered as part of the evaluation. The main goal is to ensure all mutants that were inserted by MASC are located and marked as killed or unkilled. Across the five new crypto-detectors the average number of found mutants is 14,070.

Step 3 - Identifying Flaws (RQ1): For the apps using the exhaustive approach I randomly analyzed many of mutants that went undetected to locate potential flaws. I took

the same approach as the original work and looked at misuses that the crypto-detector claims it can detect but in reality fails to detect it. To make sure all flaws were novel I exempted the exceptions that were stated specifically in the crypto-detector's documentation to ensure a fair evaluation. After all the goal is to determine flaws where crypto-detectors make claims but fail to meet those claims. Since this work was done in the original MASC the claim remains consistent that "while a crypto-detector may seem flawed because it does not detect a newer, more recently identified misuse pattern, we confirm that all the flaws we report are due to misuse cases that are older than the tools in which we find them." This is because misuses that were used to evaluate the detectors are still the most discussed misuses between 1998-2022 and since all the tools used either have new versions or were recently deployed (Amazon Code Guru). In addition to confirm flaws I used the minimal app created for the original work that contained only the undetected misuses to re-analyze the all the tools. This app was also altered to include mutations created by the new operators to verify these as well.

Step 4 - Characterizing flaws (RQ2): Flaws were grouped into the same flaw classes, based on most likely cause, used for the original evaluation. The only difference was the flaw classes were updated to include the new mutants, but the original flaw classes still remain intact. This was done to help further grasp why are the tools might fail. In addition, all 13 crypto-detectors were evaluated using the minimal examples represented by each flaw to gain a further understanding as to why a flaw might be missed despite the documentation make claims as such. Flaws were reported to their respective tool maintainers, and so far SonarQube has created five high priority issues in response to the reported flaws.

Step 5 - Understanding the practical impact of flaws (RQ3): To answer this research question for the extension I analyzed repositories to ensure that the new misuses that were introduced could also be found in real world applications. This was done using GitHub Code Search and was additionally confirmed manually.

Step 6 - Attributing flaws to mutation vs base instantiation: To ensure that a flaw that MASC found was due to a mutation and not simply a lack of coverage of the base case,

each crypto-detector was evaluated with the most basic version of the misuse to determine if it was able to catch that. An example of this would be directly testing something such as Cipher.getInstance("DES"), if this use case is caught it is then possible for it to detect mutations of this statement.

Step 7 - Reevaluation of the original eight crypto-detectors: To reevaluate the original crypto-detectors I used the same approach that was conducted on them in the original work. I ensured that they had new versions since the original work was published. To ensure that what I found was a flaw that reappeared rather than a flaw that had not been fixed from the original work, I made sure that the issue that was reported by the original authors was closed. This was to ensure that the bug was addressed. I also report if any of the flaws present in the original work that were never fixed are still present in the new version.

Chapter 5

Results and Findings

5.1 Analyzing Flaws

The new analysis of the 13 crypto-detectors results in six new flaws being found. Additionally, the analysis found 19 flaws across the 5 new crypto-detectors that were located in the original set of flaws found by MASC. For this work I included an updated version of the flaw classes with the newly located flaws and present an updated table that maps the flaws to the new versions of the crypto-detectors and new crypto-detectors. Similarly, to the original work I found that the majority of total flaws found could be attributed to mutations rather than just using the base case versions of the misuse. This continues to prove that mutation testing provides better results for finding flaws than just using the base cases alone to determine flaws.

I also found that there were two cases where flaws that were not present in the original evaluation appeared in the reevaluation of the new version. This occurred in the LGTM evaluation with F3 and CogniCrypt with F7. This shows that flaws do have a tendency to reappear and demonstrates that evaluation of crypto-detectors is not a one time thing, it needs to be ongoing. While it is clear that it is not a common occurrence for flaws to reappear it is clear from the results that it can happen. Each release has the potential to reveal more bugs and possibly bring back old misuses. In addition, across the reported flaws

a total of 58 are still present in the new versions of the crypto-detectors even after these issues were reported. However, notably seven of the eight reevaluated crypto-detectors did see improvements since the original evaluation was done. Due to the issues that were reported by the original authors this shows that MASC at least help create some improvements in the tools. QARK is the only crypto-detector that saw no change since the original work.

Since I reevaluated crypto-detectors found in the original paper I used the patched versions that included a fix for the multidex issue presented in the initial paper. However I noticed, notably with Amazon Code Guru, there does appear to be a limit to the number of recommendations it creates for a specific misuse. For example, when I used the exhaustive scope to evaluate the tool the misuse produced a "weak cipher algorithm" warning but it only reported 100 of these despite many more being seeded within each program. This would then mean after the user fixes the flaws they would have to know to run another analysis on their code. Similar to the first group this does affect the reliability of the results and makes it challenging to determine how well it evaluates many misuses. I will further discuss this fault with Amazon Code Guru and the crypto-detectors that did not fully analyze the code in the Discussion section.

FC1: String Case Mishandling (F1): This class was the motivating example found for the original work and for consistency remains in its own class for this work. This example can be found in Table 1 as F1. For this case a developer may use des or dEs (instead of the expected DES) in Cipher.getInstance(<parameter>) without any errors being raised. This was not found by Snyk despite being able to detect the base case version of this misuse. Condiga and QARK were also unable to detect this flaw but were unable to detect this base case as well.

FC2: Incorrect Value Resolution (F2 – F9): For these flaws 11/13 crypto-detectors failed to successfully detect all 9 flaws. Notably though the crypto-detectors that were evaluated in the initial paper did perform better on average than the 5 new tools that

were evaluated. Snyk and SonarQube performed on par with some of the tools that were initially evaluated.

SonarQube was able to detect some of the more simple cases but failed due to the tool not evaluating method invocations. SonarQube is only capable of detecting String literals and can detect it if it is contained in a variable.

FC3: Incorrect Resolution of Complex Inheritance and Anonymous Objects (F10 – F13): Flaws in this class occur because 10 crypto-detectors were unable to resolve the complex inheritance relationships among classes. These flaws were found specifically by using the flexible mutation operators. As found in the original paper this is clearly a consideration for some of the crypto-detectors while others did not consider this in design.

FC4: Insufficient Analysis of Generic Conditions in Extensible Crypto-APIs (F14 – F16): The flaws in this section represent the inability of crypto-detectors to identify fake conditions and also true conditionals. This determines if a crypto-detector is capable of following path sensitivity.

FC5: Insufficient Analysis of Context-specific Conditions in Extensible Crypto-APIs (F17 – F19): The flaws found within this class are similar to those found in FC4, however, the fake conditionals are contextualized to the overidden function. This would simulate the behavior of an evasive developer because one my try to add further realism to fake conidtions to avoid tools that are capbable of detecting simple generic conditions.

FC6: Newly Introduce Flaws (F20 - F25): These flaws mostly focus on interprocedural behavior and are similar to flaws found within FC2, however, they were separated to help provide more of an emphasis on these newly introduced flaws. F23 - F25 are more complicated cases of F7 and due to F7 being present in 11/13 crypto-detectors it was not surprising that none of these three flaws were found by the crypto-detectors.

F20 was the least present flaw among the newly introduced flaws where the mutations were found by 4 of the crypto-detectors. F21 and F22 were found by only two crypto-detectors and interestingly were not able to be located by the minimal case evaluation.

They were only revealed when placed within Android apps in the similarity scope for the two crypto-detectors.

Finally, with the newly introduced operators that allowed for a user specified number of method calls and nested conditionals (F7 and F25), I had planned to test a varying number of method calls and nested conditionals. I did run minimal cases with 3, 4, 5, 10, 15, and 20 method calls/conditionals. However, since the simple versions of these were not able to be located I could not fully test the limits of how many calls or conditionals the crypto-detectors actually evaluated before stopping as planned when creating these operators. As these tools improve these operators will be able to be more fully utilized and help test the limits of the crypto-detectors.

5.2 Exhaustive Results

For the 5 new crypto-detectors it is notable how many misuses SonarQube is able to identify. Out of the apps that were able to easily be run on it, it had the highest percentage of found misuses. This seems to demonstrate that SonarQube is the most thorough when analyzing files. However, running Android apps on SonarQube led to many failures as well and this is why currently the number is so low. It is currently being looked into if it will be possible to run the remaining applications.

It is also notable how well Snyk performed in this section. It misses many of the minimal examples but is capable of looking through many of the files and conducts a fairly thorough analysis. While it did miss 7,000 cases it still performed very well compared to the rest of the group. Snyk has more issues with the variety of misuses it can detect but does check every Java file for them.

It is also notable how Code Guru performed. While this is the number of reported misuses it is possible that Code Guru detected more while not reporting them. Several of the applications run reached 100 insecure cipher algorithm warnings and stopped there. It is unlikely that Code Guru happened to find exactly 100 instances in each of these

apps. This is why I believe once it reaches a certain number of a specific misuse it does not continue reporting. This makes it difficult to determine how well Code Guru looks through files.

Table 5.1: Descriptions of Original Flaws discovered by Analyzing crypto-detectors.

ID	Flaw Name (Operator)	Description of Flaws
	Flaw Class 1 (FC1): String Case Mishandling +	
F1	smallCaseParameter ($\mathbf{OP}_1$)	Not detecting an insecure algorithm provided in lower case; *e.g.*,`Cipher.getInstance("des");`
	Flaw Class 2 (FC2): Incorrect Value Resolution +	
F2	valueInVariable ($\mathbf{OP}_2$)	Not resolving values passed through variables. *e.g.*,`String value = "DES"; Cipher.getInstance(value);`
F3*	secureParameterReplaceInsecure ($\mathbf{OP}_4$)	Not resolving parameter replacement; *e.g.*, `MessageDigest.getInstance("SHA-256".replace("SHA-256", "MD5"));`
F4*	insecureParameterReplaceInsecure ($\mathbf{OP}_4$)	Not resolving an *insecure* parameter's replacement with another *insecure* parameter *e.g.*, `Cipher.getInstance("AES".replace("A", "D"));` (*i.e.*, where "AES" by itself is insecure as it defaults to using ECB).
F5*	stringCaseTransform ($\mathbf{OP}_3$)	Not resolving the case after transformation for analysis; *e.g.*,`Cipher.getInstance("des".toUpperCase(Locale.English));`
F6*	noiseReplace ($\mathbf{OP}_4$)	Not resolving noisy versions of insecure parameters, when noise is removed through a transformation; *e.g.*, `Cipher.getInstance("DE$S".replace("$", ""));`
F7	parameterFromMethodChaining ($\mathbf{OP}_5$ $\mathbf{OP}_{13}$)	Not resolving insecure parameters that are passed through method chaining, *i.e.*, from a class that contains both secure and insecure values; *e.g.*,`Cipher.getInstance(obj.A().B().getValue());` where obj.A().getValue() returns the secure value, but obj.A().B().getValue(), and obj.B().getValue() return the insecure value.
F8*	deterministicByteFromCharacters ($\mathbf{OP}_6$)	Not detecting *constant IVs*, if created using complex loops, casting, and string transformations; *e.g.*, a `new IvParameterSpec(v.getBytes(),0,8)`, which uses a `String v=""; for(int i=65; i<75; i++){ v+=(char)i;}`
F9	predictableByteFromSystemAPI ($\mathbf{OP}_6$)	Not detecting *predictable IVs* that are created using a predictable source (*e.g.*, system time), converted to bytes; *e.g.*,`new IvParameterSpec(val.getBytes(),0,8);`, such that `val = new Date(System.currentTimeMillis()).toString();`
	Flaw Class 3 (FC3): Incorrect Resolution of Complex Inheritance and Anonymous Objects	
F10	X509ExtendedTrustManager ($\mathbf{OP}_{12}$)	Not detecting *vulnerable SSL verification* in *anonymous inner class objects* created from the `X509ExtendedTrustManager` class from JCA; *e.g.*, see Listing **??** in Appendix).
F11	X509TrustManagerSubType ($\mathbf{OP}_{12}$)	Not detecting *vulnerable SSL verification* in anonymous inner class objects *created from an empty abstract class* which implements the `X509TrustManager` interface; *e.g.*, see Listing **??**).
F12	IntHostnameVerifier ($\mathbf{OP}_{12}$)	Not detecting *vulnerable hostname verification* in an anonymous inner class object that is created from an *interface that extends* the `HostnameVerifier` interface from JCA; *e.g.*, see Listing **??** in Appendix.
F13	AbcHostnameVerifier ($\mathbf{OP}_{12}$)	Not detecting *vulnerable hostname verification* in an anonymous inner class object that is created from an *empty abstract class* that *implements* the `HostnameVerifier` interface from JCA; *e.g.*, see Listing **??** in Appendix.
	Flaw Class 4 (FC4): Insufficient Analysis of Generic Conditions in Extensible Crypto-APIs	
F14	X509TrustManagerGenericConditions ($\mathbf{OP}_7$, $\mathbf{OP}_9$, $\mathbf{OP}_{12}$)	Insecure validation of a overridden `checkServerTrusted` method created within an anonymous inner class (constructed similarly as in **F13**), due to the failure to detect *security exceptions thrown under impossible conditions*; *e.g.*, `if(!(true||arg0 == null||arg1 == null)) throw new CertificateException();`
F15	IntHostnameVerifierGenericCondition ($\mathbf{OP}_8$, $\mathbf{OP}_{12}$)	Insecure analysis of vulnerable hostname verification, *i.e.*, the `verify()` method within an anonymous inner class (constructed similarly as in **F14**), due to the failure to detect *an always-true condition block that returns* `true`; *e.g.*,`if(true || session == null) return true; return false;`
F16	AbcHostnameVerifierGenericCondition ($\mathbf{OP}_8$, $\mathbf{OP}_{12}$)	Insecure analysis of vulnerable hostname verification, *i.e.*, the `verify()` method within an anonymous inner class (constructed similarly as in **F15**), due to the failure to detect *an always-true condition block that returns* `true`; *e.g.*,`if(true || session == null) return true; return false;`
	Flaw Class 5 (FC5): Insufficient Analysis of Context-specific, Conditions in Extensible Crypto-APIs	
F17	X509TrustManagerSpecificConditions ($\mathbf{OP}_7$, $\mathbf{OP}_{12}$)	Insecure validation of a overridden `checkServerTrusted` method created within an anonymous inner class created from the `X509TrustManager`, due to the failure to detect *security exceptions thrown under impossible but context-specific conditions, i.e., conditions that seem to be relevant due to specific variable use, but are actually not*; *e.g.*,`if (!(null != s || s.equalsIgnoreCase ("RSA")|| certs.length >= 314))throw new CertificateException ("RSA");`
F18	IntHostnameVerifierSpecificCondition ($\mathbf{OP}_8$, $\mathbf{OP}_{12}$)	Insecure analysis of vulnerable hostname verification, *i.e.*, the `verify()` method within an anonymous inner class (constructed similarly as in **F14**), due to the failure to detect *a context-specific always-true condition block that returns* `true`; *e.g.*,`if(true || session.getCipherSuite().length()>=0) return true; return false;`
F19	AbcHostnameVerifierSpecificCondition ($\mathbf{OP}_8$, $\mathbf{OP}_{12}$)	Insecure analysis of vulnerable hostname verification, *i.e.*, the `verify()` method within an anonymous inner class (constructed similarly as in **F15**), due to the failure to detect *a context-specific always-true condition block that returns* `true`; *e.g.*,`if(true || session.getCipherSuite().length()>=0) return true; return false;`

+ flaws were observed for mulitple API misuse cases
*Certain seemingly-unrealistic flaws may be seen in or outside a crypto-detector's "scope", depending on the perspective; see Section **??** for a broader treatment of this caveat.

Table 5.2: Descriptions of Newly discovered Flaws by Analyzing crypto-detectors.

ID	Flaw Name (Operator)	Description of Flaws
F20	StaticBytesInKeystore ($\mathbf{OP}_{19}$)	Not detecting bytes that are created using a static source converted to bytes and passed into IV; *e.g.*,`new IvParameterSpec(val.getBytes(),0,8);`, such that `byte[] val = "12345678".getBytes();`
F21	CharArrayToString ($\mathbf{OP}_{17}$)	Being unable to convert an array of Char into a String object and parsing the value; *e.g.*,`javax.crypto.Cipher.getInstance(String.valueOf(cryptoVariable));`, such that `char[] cryptoVariable = "DES".toCharArray();`
F22	UnsafeValueFromSubstring ($\mathbf{OP}_{18}$)	Not detecting a Substring misuse being parsed from a longer string; *e.g.*,`javax.crypto.Cipher.getInstance("secureParamAES".substring(11));`,
F23	ObjectSensitivity ($\mathbf{OP}_{16}$)	Not being able to differentiate between two instances of the same object type one containing a misuse and one containing a safe value; *e.g.*,`String securecipher = new CipherPack().safe().getpropertyName(); String unsecurecipher = new CipherPack().unsafe().getpropertyName(); securecipher = unsecurecipher; javax.crypto.Cipher cryptoVariable = javax.crypto.Cipher.getInstance(securecipher);`
F24	parameterBuiltFromMethodCalls ($\mathbf{OP}_{15}$)	Unable to detect the construction of a misuse String based on calls to methods that construct the misuse; *e.g.*, *e.g.*, see Listing ?? in Appendix
F25	parameterFromNestedConditionals ($\mathbf{OP}_{14}$)	Being unable to keep track of a variable as it is passed through a nested number of conditional statements; *e.g.*, see Listing ?? in Appendix

+ flaws were observed for mulitple API misuse cases
*Certain seemingly-unrealistic flaws may be seen in or outside a crypto-detector's "scope", depending on the perspective; see Section ?? for a broader treatment of this caveat.

Table 5.3: Flaws observed in different static crypto-detectors

Class	ID	AC	SQ	SY	CD	DS	NLGTM	NCG	NCC	NSB	NTX	NQA	NSL	NGCS
FC1	F1	✓	✓	✗	Ø	✓	✓	✓	✓	✓	✓	Ø	✓	✓
FC2	F2	✓	✓	✓	Ø	pr	✓	✓	✓	✓	✓	Ø	✓	✓
	F3*	✗	✗	✗	Ø	✗	✗	pr	✗	✗	✗	Ø	✗	✓
	F4*	✗	✗	✗	Ø	✗	✗	✓	✗	✗	✗	Ø	✗	✗
	F5*	pr	✗	✗	Ø	✗	✓	✓	✗	✓	✗	Ø	✓	✓
	F6*	✗	✗	✗	Ø	✗	✗	pr	✗	✗	✗	Ø	✗	✗
	F7	✗	✗	✗	Ø	✗	✓	✗	✗	✗	✗	Ø	✗	✓
	F8	Ø	✓	✗	Ø	Ø	✗	✗	✓	✓	✓	Ø	✓	Ø
	F9	Ø	✓	✗	Ø	Ø	✗	✗	✓	✓	✓	Ø	✓	Ø
FC3	F10	Ø	✓	✓	Ø	✓	✗	✗	-	✓	✓	pr	✓	Ø
	F11	Ø	✓	✓	Ø	✗	✗	✗	-	✓	✓	pr	✓	Ø
	F12	Ø	✓	✗	Ø	✓	✗	✗	-	✓	✓	-	✓	Ø
	F13	Ø	✓	✗	Ø	✗	✗	✗	-	✓	✓	-	✓	Ø
FC4	F14	Ø	✗	✓	Ø	✗	✗	✗	✓	✓	✓	✗	✓	Ø
	F15	Ø	✗	✗	Ø	✗	✓	✗	-	✓	✓	-	✓	Ø
	F16	Ø	✗	✗	Ø	✗	✓	✗	-	✓	✓	-	✓	Ø
FC5	F17	Ø	✗	✓	Ø	✗	✗	✗	✓	✓	✓	✗	✓	Ø
	F18	Ø	✗	✗	Ø	✗	✓	✗	-	✓	✓	-	✓	Ø
	F19	Ø	✗	✗	Ø	✗	✓	✗	-	✓	✓	-	✓	Ø
FC6	F20	✗	✓	✓	Ø	✗	✗	✗	✓	✗	✗	✗	✓	✗
	F21	✗	✗	✗	Ø	✗	✓	✗	✗	✗	✗	✗	✗	✓
	F22	✗	✗	✗	Ø	✗	✓	✗	✗	✗	✗	✗	✗	✓
	F23	✗	✗	✗	Ø	✗	✗	✗	✗	✗	✗	✗	✗	✗
	F24	✗	✗	✗	Ø	✗	✗	✗	✗	✗	✗	✗	✗	✗
	F25	✗	✗	✗	Ø	✗	✗	✗	✗	✗	✗	✗	✗	✗

✗ = Flaw Present, ✓ = Flaw Absent, `pr` = Flaw partially present, -= detector does not claim to handle the misuse associated with the flaw, Ø= detector claims to handle but did not detect base version of misuse nr = not yet run; AC = Amazon Code Guru, SQ = SonarQube, SY = Snyk, CD = Codiga, DS = DeepSource, NCG = CryptoGuard version 04.05.03, NCC = CogniCrypt version 2.7.3, NSB = SpotBugs with FindSecBugs version 1.12.0 , NTX = Coverity version 2022.6.0 , NQA = QARK version 4.0.0 , NSL = ShiftLeft version 2.1.1, NGCS = GitHub Code Security version 2.12.6, NLGTM = LGTM version 2.12.6 .
*Certain seemingly-unrealistic flaws may be seen in/outside a crypto-detector's "scope", depending on the perspective; see Discussion for a broader treatment of this caveat.

Table 5.4: Mutants analyzed vs detected by crypto-detectors

Tool	Input Type	Analyzed	Detected
Snyk	Java Src Code	47,002	40,877
DeepSource	Java Src Code	47,002	17,028
Codiga	Java Src Code	26,725	0
SonarQube	Java Src Code	13,749	11,601
Amazon Code Guru	Java Src Code	46,967	840
CryptoGuard	`apk` or `jar`	20,303	18,616/19,759
CogniCrypt	`apk` or `jar`	20,303	475
$Tool_X$	Android or Java Src Code	20,303	48
SpotBugs	`jar`	17,788	17,715
QARK	Java Src Code or `apk`	46,324	0
LGTM	Java Src Code	34,846	21,474
GCS	Java Src Code	34,846	21,440
ShiftLeft	Java Src Code	46,252	35,200

Chapter 6

Conclusion

MASC has been demonstrated once again as being effective at finding flaws in crypto-detectors. The work has been greatly expanded but there are still more areas for further improvement. This new extension of MASC has also reintroduced some areas of discussion initially brought to light by the original work. Since some time has passed some of the perspectives of these discussions have shifted.

6.1 Limitations

MASC is still designed to help find flaws in crypto-detectors and still cannot gurantee all flaws in a crypto-detector will be found. In fact formal verification should still be done in conjunction with MASC. MASC's design from the very begining was to allow for "systematic evaluation of crypto-detectors, which is an advancement over manually curated benchmarks." This still remains true while the extension has expanded MASC's functionality and coverage it is still not representative of all misuses. MASC is still held back by the following limitations:

1. Completeness of the Taxonomy: The same approach was used to ensure the taxnomy was comprehensive as the original work. All steps were performed carefully and utilized the same best pratices found in other work. However, it is still possible that some cases or sources could be missed during extraction. The extra evaluation step was added to help

ensure accuracy but it is still possible that some subtle contexts were once again missed. Up to current date though I believe that this is still, to the best of my knowledge, the most comprehensive found in recent works taxonomy in this space.

2. Focus on Generic Mutation Operators: This was a concern of the original paper since the goal was to apply as many misuses from the taxonomy as possible. This was a main area of expansion for this extension. MASC now contains many more new operators that do represent some of the more specific cases found in the taxonomy. However, the taxnonmy is still not fully represented and many of the new operators were still designed to have multiple use cases. Priorities for new operators were still focused on covering more potential misuse even though the focus was exclusivly on expanding the restrictive operators.

3. Focus on Java and JCA: MASC's approach is still informed by JCA and Java. No work has been done since the original paper to adapt MASC to other languages at this time.

4. Evolution of APIs: Since changes are still made to how JCA operates this may eventual lead to changes being necessary in MASC. Up to now MASC is current and the operators from the original paper do still function correctly. However, as time passes some changes may become necessary despite MASC using reflection and automtated code generation to ensure flexibility. In addition it is still an ongoing project to include more misuse cases within MASC to get closer to fully representing the current taxonomy and beyond.

5. Relative Effectiveness of Individual Operators: My research did not look into this limitation that was present in the original paper. This paper looks further into what MASC is capable of doing as a whole but does not evaluate how effective each operator is at finding flaws. This still requires its own study.

6. Consistency with the Original Work: I was in constant communication with the original authors and confirming with them how the original work was created. I inncorportated feadback from them and had them review changes to the tool to help ensure

as much consistency as possible. In addition for evaluation and expansion I followed all the steps laid out by the original work to ensure that the work could stand as one whole project rather than an add on. I believe that this was done to the best of my ability and MASC now stands as an expanded and more comprehensive tool. Since I was not directly involved in the original work there may be some small details that were not done exactly the same as the original work.

6.2 Discussion

Security-centric Evaluation Design: MASC still places a heavy focus on security centric design. From the perspective of this framework it is believed that security should come first no matter how unlikely or evasive a case may be. This is part of why some of the new operators were designed to create more evasive cases. This idea still clashes with the idea that designers of tools look more into a technique-centric perspective. Many of these tools are not designed with a threat model in mind or directly from a security prespective. In fact many of the tools analyze best coding practices in addition to looking for security misuses. This leads to a gap between what is expected of a tool that claims it can perform security checks.

What is scope for Technique-centric Design: There is still an ongoing discussion on the space on what the scope should be. Should there be more of a focus placed on what is most common and uncommmon or should it be on what can easily be computed statically vs cannot easily be computed. Even from a security prespective this is challenge since common misuses would be exptected to be caught, however, new misuses pose a bigger threat due to the lack of awareness of the looming threat. Optimially most tools would like to cover all possible cases but since this is not feasible the debate is what is most important and what should be expected on crypto-detectors.

The Need to Strengthen Crypto-Detectors: Many detectors make claims with the assurance that they can detect certain misuses. When tested many of these claims are

proven to still fall short of expectations. If crypto-detectors claim to be able to secure your code it is important that they actually can. This means detecting uncommmon cases and being held accountable for falling short. MASC has shown that it is possible to find gaps in crypto-detectors and help them improve but they are still a long way off from being able to detect hard to statically compute misuses cases. This leads to the question of should crypto-detectors be able to make the claims they do to secure your code?

Shifting toward security centric design: The original work showed that there is interest in developers to make tools more secure. This was something they strived for and it was proven that some of the tools that were evaluated did improve since the first paper. This demonstrates that not only did the express interest they demonstrated interest. Even in the new tools that were evaluated when issues were reported to them they were eager to make a fix to improve their tool. Many developers have expectations that these tools help make their code secure and a lot of times are not aware of common misuses. The further these tools can expand with security centric design in mind the more developers can expect to rely on them.

Pushing toward SARIF: As mentioned SARIF is a relatively new SAST output format. Some crypto-detector designers have been reached out to and expressed interest in being able to produce this format for their tool. MASC integrated this in hopes of helping to encourage more crypto-dectector designers to look into this format. If there is a standard output not only does it make it easier to evaluate how well tool perform it also allows for new tools to be produced to help users evaluate their applications. Having a standard output as an expectation can lead to easier evaluation and can help create a way to fully automize the evaluation of crypto-detector. This would help users be able to easily know how reliable any tool is before they use it. In addition if this was a standard output this would also allow tools to more easily report all misuses found. As mentioned in a prior section some crypto-detectors such as Amazon Code Guru do not provide a full report for their analysis. This is likely due to UI limitations since the whole codebase is still scanned. Having an output in the style of SARIF can help provide a full report to the

user since it likely would not have to be directly consumed by the user, it would be parsed and display the output that way.

The Need for constant evaluation: As shown in the reevaluation of the original crypto-detectors some bugs or flaws have the potential to reappear in future versions of a tool. Due to the ongoing iteration of crypto-detectors it is important for them to be reevaluated. While obviously an evaluation would be necessary when a new misuse appears reevaluating the old misuses is equally as important. Bugs have a tendency to reappear while refactoring and making changes and as found the same thing goes for flaws. Having a constant evaluation will help ensure that once a flaw is found and eliminated that it stays that way.

6.3 Lessons Learned

Throughout my time working on this project I went from someone who had never worked on a large software engineering project to someone conducting research and greatly expanded on the project. Throughout my time I have learned a lot about how to conduct research in a methodical way and what that means in the field of Software Engineering. I have faced many challenges throughout my work but each one has allowed me to become both a better developer and researcher. Each challenge has pushed me to obtain a new skill set. For this project I started small by simply trying to understand the codebase I was working on. As time went on and I got a further grasp I was able to build new tools such as the automated analysis. As time pushed further I was tasked with conducted research and producing my own ideas for ways to improve MASC this led to the addition of new operators and sensitity evaluator. This project has truly helped me grow as an individual.

6.4 Conclusion

The creation of a tool like MASC has helped shift the mindset of the crypyto-detectors designers into looking further into a more security-focused design. They have made claims

and as found in this extension they have improved in some areas. There is still a long way to go and by reaching out to the designers and reporting flaws I have seen that there is interest in ensuring security. With the MASC framework existing and being known to the crpyto-detector makers it is possible for them to easily perform self evaluation and continue to improve. As MASC continues to expand in the future it is possible to help push crypto-detectors into become more secure as well. With MASC existing in the same ecosystem as crypto-detectors it is possible to help push towards a more security centric design for tools and one day users can truly expect when their code is analyzed that it is secure.

Appendix A

Appendix A

A.1 Code Snippets

Listing A.1: Method Chaining ($\mathbf{OP}_5$).

```
1    Class T { String algo="AES/CBC/PKCS5Padding";
2    T mthd1(){ algo = "AES";  return this;} T mthd2(){ algo="DES";
         return this;} }
3    Cipher.getInstance(new T().mthd1().mthd2());
```

Listing A.2: Predictable/Non-Random Derivation of Value ($\mathbf{OP}_6$)

```
1    val = new Date(System.currentTimeMillis()).toString();
2    new IvParameterSpec(val.getBytes(),0,8);}
```

Listing A.3: Exception in an *always-false* condition block ($\mathbf{OP}_7$).

```
1    void checkServerTrusted(X509Certificate[] x, String s)
2    throws CertificateException {
3    if (!(null != s && s.equalsIgnoreCase("RSA"))) {
4         throw new CertificateException("not RSA");}
```

Listing A.4: False return within an *always true* condition block ($\mathbf{OP}_8$).

```
1    public boolean verify(String host, SSLSession s) {
2      if(true || s.getCipherSuite().length()>=0)}
```

```
3          return true;} return false;}
```

Listing A.5: Implementing an Interface with no overridden methods.

```
1     interface ITM extends X509TrustManager { }
2     abstract class ATM implements X509TrustManager { }
```

Listing A.6: Inner class object from Abstract type ($\mathbf{OP}_{12}$)

```
1     new HostnameVerifier(){
2       public boolean verify(String h, SSLSession s) {
3         return true; } };
```

Listing A.7: Anonymous Inner Class Object of `X509ExtendedTrustManager` (??)

```
1     new X509ExtendedTrustManager(){
2       public void checkClientTrusted(X509Certificate[] chain, String a)
            throws CertificateException {}
3       public void checkServerTrusted(X509Certificate[] chain, String
            authType)throws CertificateException {}
4       public X509Certificate[] getAcceptedIssuers() {return null;} ...};
```

Listing A.8: Specific Condition in `checkServerTrusted` method (??)

```
1     void checkServerTrusted(X509Certificate[] certs, String s)
2        throws CertificateException {
3      if (!(null != s || s.equalsIgnoreCase("RSA") || certs.length >=
           314)) {
4        throw new CertificateException("Error");}}
```

Listing A.9: Anonymous Inner Class Object of An Empty Abstract Class that implements `HostnameVerifier`

```
1     abstract class AHV implements HostnameVerifier{} new AHV(){
2       public boolean verify(String h, SSLSession s)
3           return true;}};
```

Listing A.10: Anonymous inner class object with a vulnerable `checkServerTrusted` method (F13)

```
1  abstract class AbstractTM implements X509TrustManager{} new
       AbstractTM(){
2    public void checkServerTrusted(X509Certificate[] chain, String
       authType) throws CertificateException {}
3    public X509Certificate[] getAcceptedIssuers() {return null;}}};
```

Listing A.11: Anonymous Inner Class Object of an Interface that extends `HostnameVerifier`

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

Listing A.12: Misuse case requiring a trivial new operator

```
1  KeyGenerator keyGen = KeyGenerator.getInstance("AES");
2  keyGen.init(128); SecretKey secretKey=keyGen.generateKey();
```

Listing A.13: CryptoGuard's code ignoring names with "android"

```
1  if (!className.contains("android."))
2      classNames.add(className.substring(1, className.length() - 1));
           return classNames;
```

Listing A.14: Generic Conditions in `checkServerTrusted`

```
1  if(!(true || arg0==null || arg1==null)) {
2      throw new CertificateException();}
```

Listing A.15: Transformation String formation in Apache Druid similar to **F2** which uses `AES` in `CBC` mode with `PKCS5Padding`, a configuration that is known to be a misuse [**? ?**].

```
1  this.name = name == null ? "AES" : name;
2  this.mode = mode == null ? "CBC" : mode;
3  this.pad = pad == null ? "PKCS5Padding" : pad;
4  this.string = StringUtils.format(
5      "%
```

Listing A.16: Iterative Method Chaining

```
1         Class T {
2         int i = 0;
3         cipher = "AES/GCM/NoPadding";
4         public void A(){
5             cipher = "AES/GCM/NoPadding";
6         }
7         public void B(){
8             cipher = "AES/GCM/NoPadding";
9         }
10        public void C(){
11            cipher = "AES/GCM/NoPadding";
12        }
13        public void D(){
14            cipher = "AES";
15        }
16        public String getVal(){
17            return cipher
18        } }
19
20
21        Cipher.getInstance(new T().A().B().C().D().getVal() ) ;
```

Listing A.17: Iterative Conditionals

```
1             Class T {
2             int i = 0;
3             cipher = "AES/GCM/NoPadding";
4             public void A(){
5             if ( i == 0){
6                 if (i == 0){
7                     if(i == 0){
8                         cipher = "AES";
9                     }
10                    else{
```

```
11                          cipher = "AES/GCM/NoPadding";
12                      }
13                  }
14                  else{
15                      cipher = "AES/GCM/NoPadding";
16                      }
17              } else{
18                  cipher = "AES/GCM/NoPadding";
19
20              }}
21
22              public String getVal(){
23                  return cipher
24              } }
25
26
27              Cipher.getInstance(new T().A().getVal() ) ;
```

Listing A.18: Iterative Conditionals

```
1               Class T {
2               int i = 0;
3               cipher = "AES/GCM/NoPadding";
4               public void A(){
5               if ( i == 0){
6                   if (i == 0){
7                       if(i == 0){
8                           cipher = "AES";
9                       }
10                      else{
11                          cipher = "AES/GCM/NoPadding";
12                      }
13                  }
14                  else{
15                      cipher = "AES/GCM/NoPadding";
```

```
16                      }
17              } else{
18                  cipher = "AES/GCM/NoPadding";
19
20              }}
21
22              public String getVal(){
23                  return cipher
24              } }
25
26
27              Cipher.getInstance(new T().A().getVal() ) ;
```

Listing A.19: Method Builder

```
1                   Class T {
2                   int i = 0;
3                   cipher = "AES/GCM/NoPadding";
4                   public String A(){
5                       return "D";
6                   }
7                   public String B(){
8                       return "E";
9                   }
10                  public String C(){
11                      return "S";
12                  }
13                  public void add(){
14                      cipher = A() + B() + C();
15                  }
16                  public String getVal(){
17                      return cipher
18                  } }
19
20
```

```
21                  Cipher.getInstance(new T().add().getVal() ) ;
```

Listing A.20: Object Sensitive, using the object created in Listing A.1

```
1                   T secure = new T();
2                   T insecure = new T().mthd2();
3                   secure = insecure;
4                   Cipher.getInstance(secure.getVal());
```

Listing A.21: Build Variable

```
1   String cryptoVariable = "AES";
2   char[] cryptoVariable1  = cryptoVariable.toCharArray();
3   javax.crypto.Cipher.getInstance(String.valueOf(cryptoVariable1));
```

Listing A.22: Substring

```
1   javax.crypto.Cipher.getInstance("secureParamAES".substring(11));
```

Listing A.23: Static Keystore

```
1   byte[] cryptoTemp = "12345678".getBytes();
2   javax.crypto.spec.IvParameterSpec ivSpec = new javax.crypto.spec.
        IvParameterSpec.getInstance(cryptoTemp,"AES");
```