Clever Thesis Title

Student Full Name

Hometown, HomeState, HomeCountry

Bachelor of Arts, University Name, 2013
Master of Science, University Name, 2015

A Dissertation presented to the Graduate Faculty
of the College of William & Mary in Candidacy for the Degree of
<Insert Degree Name>

Department of Computer Science

College of William & Mary
August 2018

APPROVAL PAGE

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

<Insert Degree Name>

Student Shortened Name

Approved by the Committee, August 2018

Committee Chair
Associate Professor FirstName LastName, Computer Science
College of William & Mary

Associate Professor FirstName LastName, Computer Science
College of William & Mary

Assistant Professor FirstName LastName, Computer Science
College of William & Mary

Assistant Professor FirstName LastName, Computer Science
College of William & Mary

Professor FirstName LastName, Computer Science
The University of Texas at Dallas

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

Awesome Abstract paragraph 1.

Awesome Abstract Paragraph 2.

Awesome Abstract Paragraph 3.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

First Acknowledgement Paragraph

Second Acknowledgement Paragraph

Third Acknowledgement Paragraph

Fourth Acknowledgement Paragraph

Insert heartfelt dedication here...

LIST OF TABLES

LIST OF FIGURES

Clever Thesis Title

Chapter 1

Introduction

Cryptography is esential in making sure data is confidential and secure in the modern world. Making sure cryptography primiatives are used correclty, however, has always been difficult problem to solve wether it be in critical systems like banking or the wide spread misuse of cryptographic API found in mobile and web apps. These misuses can lead to vulnerabilities that allow for data leaks and cause sensitive data to be leaked. To solve this problem researchers have designed tools that can be integrated into the development pipeline that catch these misuses and allow developers to catch the problems before they are released 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-dectors are not without their own faults. Crypto-dectors see widespread use across IDEs, corporate testing suites, by source control platforms (such as GitHub), and comercial use. Crypto-dectors 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-dectors 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-dectors beyond simple manually created benchmarks. As described in the original paper [insert citation] MASC is "the first systematic, data-driven framework that leverage the well-founded appraoch of Mutation Analysis for evaluating Static Crypto-API misuse detectors." MASC is used by a user in a imilar manner to typical mutation analysis. MASC is capable of "mutating Android/Java apps by seeding them mutants" containing crypto-API misuse. The mutated apps can then be anaylzed by a crypto-dector and seeing which mutants are not located during the analysis reveals both design and implementation flaws in that crypto-dector. For my thesis we 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. We expanded MASC in both depth and width. The updated MASC framework was also used to reevaluate Crypto-Dectors that were evaluated by the original paper and evaluate 5 additional crypto-dectors. With the new additions to MASC and new evaluation we were able to determine: do misuses that were reported and fixed have a tendency to reappear in future builds, have crypto-dectors improved since the original paper, can MASC discover new flaws in crypto-dectors, what are the chracterisitcs of these flaws, and what is the impact of the flaws on the effectiveness of crytpo-dectors 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 taxnonoy of crypto-API misuse cases containing 105 cases. These were found using a data driven process that systematially identified, studiad, and extacted 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 ... more papers and ... more misuses. MASC also had several Crypto-Mutation Operators and Scopes. These different abstractions were designed to allow flexible in MASC to create a large variety of feasible mutants. The threat model MASC is based on consists of three types of conditions that crypto-dectors are likely to face. In addition, MASC also consisted of usage base mutation operatos which were general operatos 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 overiding its verify method, with any content. The original set of operators consisted of 12 operators (6 restrictive and 6 flexible). For my thesis we focused on expanding the amount of restrictive operators since more are necessary to fully represent the taxonomy. I added ... number of 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 our T3 threat model of an evasive developer since the original work was intentially conservative with what was considered evasive. After more research it was determined that new operators could be created to more evasive and still fair to crypto-dectors. MASC also includes 3 types of mutation scopes that allow for seedinig of mutants variable fiedelty 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 runner <change name>. The automated analysis provides MASC with a way to automatically evaluate a crypto-dectors output and determine which misuses were caught or not. It leverages SARIF files (an file type similar to JSON) to determine where misuses were. It is also designed to catch flaws with crypto-dectors based on output. It can test a variety of cases against a crypto-dector 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-dectors and ensure accuracy. The other main new feature introduced as a part of MASC is the sensitity runner. This tool defined the different types of sensitivities common to crypto-dectors: flow, object, context, path, and alias sensitity. 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 sensitity 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-dector. For the evaluation of crypto-dectors I used the same methodology as the original paper but extended the number of operators and the number of applications. I evaluated 8 of the major crypto-dectors used in the original paper and also evaluated 5 addition crypto-dectors. The original paper also evaluated the crypto-dectors 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. Additionally, our findings were disclosed to the designers/maintainers of the tools.

1.1.1 Motivation and Background

The motivation for extending MASC still comes from the same idea as the original motivation behind MASC, "Insecure use of cryptographic APIs is the second more common cause of software vulnerabilities after data leaks. Many developers are likely to use crypto-dectors 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-dectors and their ability to locate misuses directly impacts the security of the end user and their data. Evaluating crypto-dectors 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 evaluting crypto-dectors. The crypto-dectors evaluated in the original paper needed to be reevaluated because bugs that get fixed have a tendency to reappear in future patches. This is a common occurence in the field of software engineering.[citation] 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-dectors 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 and SonarQube. I wanted to ensure MASC conducted an evaluation on as many crypto-dectors 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-dectors. The motivating example for this new compoent 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-dector known for being able to detect security vulnerabilities found in software prior to release. Alice had reported that their was flaw in the crypto-dector 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-dector 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-dector. DES is the common version of the misuse, however, Java supports both the uppercase and lowercase versions of this misuse. A crypto-dector 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-dectors to uderstand how the position their tool and what they claim to be capable of. When evaluating the new tools we looked through their documentation as well and ensured that they made similar claims as the originally evaluated crypto-dectors to ensure the model was consitend across all evaluated crypto-dectors. 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 5 of the newly evaluated crypto-dectors make similar claims of security assurance. Just like the original 9 crypto-dectors 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 claim what they say the can. For our threat model we continued off the same model used in the original paper. It assumed that there are some circumstances where they might be deployed in adverserial 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 3 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-dector. T2 Benign developer, harmful fix - This scenraio 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 miuse 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 intentially trying to evade the crypto-dector. When the alert from a detector is recieved 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 conditions are meant to mimic how crypto-dectros have to operate in practice and the evalution is designed based on what the crypto-dectors 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 truly ensure that all examples were accurate to cases that could be found. 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.

Chapter 2

The MASC Framework

2.1 Overview

The original work proposed a framework for Mutation-based Analysis of Static Crypto-miuse detection techniques (or MASC). [Add Fig 1] shows the design of the MASC framework as described in the original work. The framework remains the same and all the ideas from this orignal 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 the misuse cases to be grounded based on what is seen in practice. This both allows for a more focused design space as well as helps justify the misuses MASC puts into practice. For this extension this 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 orignal 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-characteristicsof cryptographic APIs. These can then be leveraged to design "general, usage based mutation oprators." These oprators being designed in this way allows for a single oprator 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 of the oprators previously designed for MASC were already capable of expressing these misuses with little to no effort. A substatial amount of work was put into extending operators to cover misuses that were not already represented in both the original and extended taxonomy. By instantiating mutants 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-dector targeted for evaluation. Based on the results produced by the target it is posible 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

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. 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 a two different components. In order to locate academic and research papers related to Crypto-API misuse, it was first necessary to 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. Specifically, these terms are shown in Figure .

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 make 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 above in Figure , 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-dectors use CWE a 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. We searched 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 matter 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.

After compiling the master list of misuse sources, a misuse case extraction was performed. We had 2 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 3 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 us to look at each individual paper. Based on the updated taxonomy that was produced we 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. If it was 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 seperate parts.

By the end of this process 13 new misuse cases were identified 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. The new misuse use cases include:

- Storing sensitive data in Java String (5 occurrences)
- Key resuse in stream ciphers (1 occurrence)
- Use of expired keys (1 occurrence)
- No clearPassword call after using PBEKeySpec (1 occurrence)
- Using AES-CTR (1 occurrence)
- Weak algorithm for password-based encryption - PBKDF1 (1 occurrence)
- HMAC for TLS with MD5 (1 occurrence)
- Using RC5* (1 occurrence)

- Using ARCFOUR (1 occurrence)
- PBEWithMD5AndDES (1 occurrence)
- Hashing Credentials - SHA-224 (1 occurrence)
- Reusing IVs & key pairs (1 occurrence)
- Manually changing hostname verifier (1 occurrence)

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 tradeoff 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 expessive of multiple misuse cases to allow for more coverage. Design of operators is also guieded 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 oprators 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 amoungst misuses of crypto-API. The restrictive operators are operators where a developer can only instantiate certaion 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 utilie 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 posibilities 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 (**??**) 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.

$\mathbf{OP}_{13}$: ***Iterative Method Chaining –*** Similar to in MASC $\mathbf{OP}_5$ (Method Chains) this operator implements method chaining to hide the value. Where this operator differs from $\mathbf{OP}_5$ is that it can take a value specified by the user and create that many method calls. The method calls are then chained in sucession. Every method call will has 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 (**??**) developer.

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.

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 (**??**) behavior and is not something a developer would accidentally do.

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. This behaviour also fits under our T3 developer.

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 (**??**) 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.

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 (**??**) 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 miuse that could slip by.

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.

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-dector is even capable of detecting misuses in the most plain cases. These mutants are seeded in simple Java or Android templates at the begining of the main method. This ensures that the mutants are found and analyzed. This is meant to simulate the behavior of a beniegn (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 miuse that is known to be detected by the operator and is used to determine how thorough the crypto-dectors when analyzing files. This scope is meant to analyze a T3 developer they may hide miuses 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 beniegn developers (T1 and T2) would potentially place misuses and is meant to test how well the crypto-dectors analze 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 miusse cases, and (3) seeding/inserting the instantiated mutants in Java/Android source code at targeted locations." The extension of the framework was build upon the foundation that was previously laid. All additions remained consistent with the implementation of the original work. 1. Selecting misuse cases from the Taxonomy: In the original work 19 misues 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 prevelant cases appeared as well. For the extened 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 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. Implenting mutants: The mutation operators described in the previous chapter along with the original 12 operators were desgined to be applied to one or more crypto-API, for instatiating 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 syntatic requirements of each API that is being implemented (such as the requirement of surrounding try-catch block with appropriate exception handling) and the semantic rrequirement of the specific misuse that is being instantiated. MASC used Hava 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 complilable 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-dector with little effor from the user. 3. Indentifying 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 depdencies 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 uncessary parts 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 insterted 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 compliation 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

3.0.1 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.

3.0.2 Sensitivity Evaluator

The sensitivity evaluator is a new component of the MASC framework designed as an alternitive way for users evaluate crypto-dectors. This tool is designed to help users access the oprators 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 tools are built with these sensitivities in mind and claim to be able to logically handle some of them. The main sensitivities I found when look 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 a extremely expensive computationally.

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 1 and we pass this into a method this passed value would be an alias of x.

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.

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 - Field sensitivity is 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 Sensitivty - 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 that there is not vulnerability taking place.

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 under 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 evaluting 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. 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 of 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 the research.

3.0.3 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. This 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 so I leveraged 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 properties file used for mutation. Using this information the tool parses through the MASC properties 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 properties file. It was made possible that when MASC was run it could run the output it creates against a Crypto Detector 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-dectors 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.

3.0.4 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 mutated 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

Chapter 5

Results and Findings

Chapter 6

Conclusion

6.0.1 Limitations

6.0.2 Discussion

6.0.3 Conclusion

Appendix A

First Appendix with Awesome Data

A.1 Awesome Appendix Section