Reproducing: Inferring Crypto API Rules from Code Changes

Analyzing the taken approach, and discuss possible improvements

Raphael Jenni
OST Eastern Switzerland University of Applied Sciences
Supervised by Prof. Dr. Luc Bläser
FT 2021

Abstract

Analyzing big code to gain information to help a developer get insights is a long-standing topic. With the help of big code, the possibilities have increased dramatically. The paper 'Inferring Crypto API Rules from Code Changes" [PZT+18] attempts to use big code to detect crypto API usage changes and derive rules from them. This paper aims to reproduce its results, analyze its approach and provide directions on building upon this idea.

To do that, we recap the paper's approach and the results. We then report on the findings and issues while implementing the described approach. The overall impression is that the paper is a good starting point for further research but is not applicable for a real-world application with a large codebase. The reproduced results are documented, and problems are discussed. We end the paper with three rough ideas that are based on the paper's approach: (i) "Enhance Code Change Detection" where the code change detection method should be extended to handle more cases of code changes. (ii) "End-to-end Toolchain" where a toolchain is proposed to introduce machine learning to have a system that detects errors based on code changes from open source projects. (iii) "Currently-Changing Dashboard" where a dashboard is proposed that shows currently changing API usages/upgrades, vulnerability and security issue fixes, and other changes to a developer and also delivers insights on what parts of a project need to be investigated.

Keywords: Code Analysis, Change Detection, Big Code

1 Introduction

This paper aims to reproduce the approach and findings of the paper "Inferring Crypto API Rules from Code Changes" [PZT+18]. It will start with a short recap of the different approaches taken. It then continues with the reproduced results, including all the findings made during the implementation. It concludes in the last section with a summary of the results. Furthermore, based on the paper under investigation, possible next steps are proposed.

2 Paper Recap

The paper "Inferring Crypto API Rules from Code Changes" [PZT+18] describes an approach where changes from a version control system get analyzed to find common Java Crypto API usage changes. It does that by farming repositories from GitHub, GitLab, and other sources, that contain usages of the Crypto API. Those repositories are then converted into versions for each file. Each version gets compared with the one next in line, and an abstract representation is built. This representation then builds the basis for filtering out unimportant changes that do not change anything of the functionality like renamings or refactorings. The relevant changes then get clustered by similarity. It is now the job of a human to derive the rules from the clusters.

The following section summarizes the different sections of the paper and describes the most important aspects of it to create a baseline for the reader.

2.1 Code Abstraction

For applying static analysis, the code first needs to be parsed. The parsing process is rather generic and nothing special. The exciting part comes with reducing details by abstracting the parsed AST into a DAG for each object present in the code. Other than the AST, this DAG is independent of the code structure. Inserting a line that, for example, prints out some information does change the AST, but not the DAG. As long as the order of the invocations is the same, the DAG stays the same. Furthermore, the DAG focuses only on the usages of a single object instead of the whole code structure.

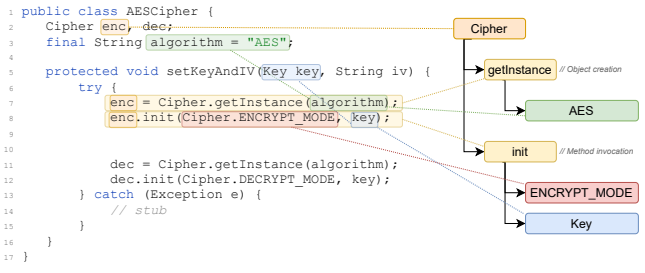

Figure 1. Conversion of Code into an abstract DAG [PZT+18].

As visualized in Figure 1, the graph represents usages of the field `enc` of type `Cipher`. The DAG also includes all the used parameters when interacting with the object.

2.2 Changes Paring

When having constructed all the DAGs for both versions, the connection between each DAG from version 1 to version 2 is not clear (see Figure 2). Therefore, a mechanism for

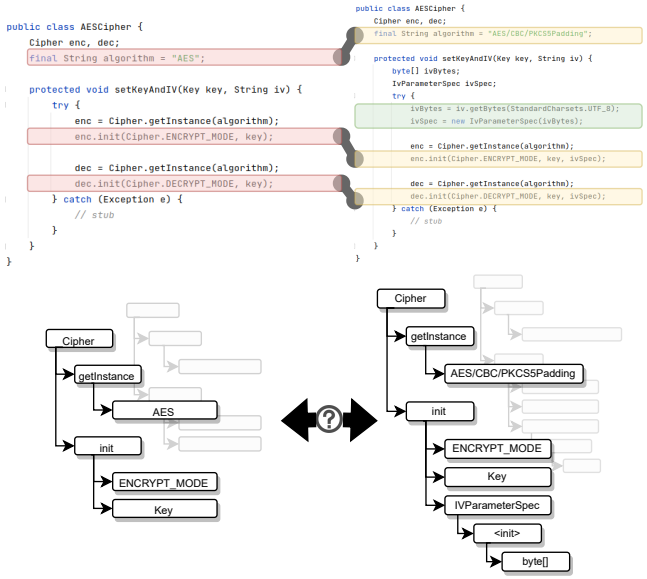

Figure 2. Analyzing the Code results in multiple DAGs. How to find the matching ones [PZT+18]?

matching DAGs is needed. There is no bulletproof way to do this, but the paper uses a method with a very high success rate. This pairing is done by calculating the similarity of all DAG combinations and selecting the match configuration that results in the lowest overall score. This similarity is calculated by taking the ratio of common nodes over the total number of nodes.

2.3 Usage Changes

After having all the matched DAGs, the usage changes can be investigated (see Figure 2). Extracting usage changes is done by creating all paths along the DAGs and extracting the longest path prefix. This prefix enables us to treat the rest of a path with this specific prefix as a change. The process is visualized in Figure 3. After having all usage changes, a filtering step is executed to only look at the changes with real usage changes and ignore changes like refactorings, introducing a new object, or removing an object. The filtering works by looking at the added and removed paths. If no paths were added or removed, then no changes happened. If paths were added but non were removed, the change introduced a new API usage. If paths were removed but non were added,

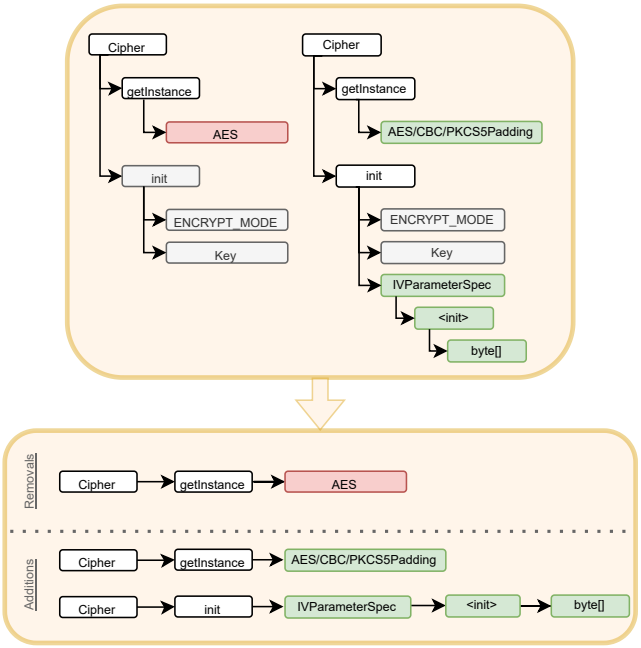

Figure 3. Usage Changes Extraction Process [PZT+18].

the change removed an API usage. If the same path is present multiple times, then the change is already covered by a prior path and gets therefore ignored. Those four filtering steps clear out most of the changes. When all the filters are applied, only the relevant changes remain.

2.4 Usage Changes Clustering

At this point, the usage changes are clustered with an agglomerate hierarchical clustering algorithm. The clustering is done by putting all changes into the leaves of a binary tree structure. The two most similar changes, meaning changes with the nearest usage distance, get combined into a node one level higher in the tree. This process is then done until the tree is only one level deep. It is then the person's job, interested in the changes, to decide which level in the tree to look at. The paper extracted 13 rules regarding security issues for the Java Crypto API. An example for such a rule would be: "Use SHA-256 instead of SHA-1".

3 Reproduced Results

After recapping the paper's approach, this section will cover the reproduced results together with the challenges and problems faced.

There were two main issues when reproducing the results in the paper. First, no code has been made publicly available. Second, the approaches taken by the authors were described mainly in a very general and brief manner. Those obstacles led to having to make many assumptions which in

turn affected the results and their comparability. Nevertheless, all code written is based on best effort for the available information and time available.

3.1 Code Abstraction

The original code was written in Python. For ease of use when handling Java code, the rebuilt version is written in Kotlin and leverages the OpenJDK parser.

The OpenJDK parser is a private API, so it is not recommended for general use. The private nature also means that no documentation is available. Nevertheless, after understanding its internals, the parser works excellent and provides an easy and quick way to parse Java code. Based on the parsed AST, the code abstraction could be derived.

Re-implementing the code abstraction described in the paper was possible but is most likely not complete. Also, in the end, only parts of the initially defined abstractions were used to create the DAGs. For the abstraction part, the examples presented in the paper could be reproduced. Other code with a simple and flat structure also works relatively reliable. The abstraction falls apart as soon as the code gets more complicated or deeply nested.

Examples of cases where the abstraction is not specified enough, and therefore are ignored or not explicitly handled are described subsequently.

Nested method invocations. The code snipped `example.methodInvocation(asset.getWidth(), asset.getHeight());`, consists of a method invocation on "example" (`.methodInvocation`) and two method invocations on "asset" (`.getWidth` and `.getHeight`). Based on the paper, it is not clear if the DAG of `method` contains two parameters of type `Int` or if it includes the method invocations and their DAG as well. Furthermore, it most likely is different for methods that return an object and methods that return a primitive type. In the rebuilt version, for simplicity and consistency, the method's return type is taken. In the case of the previous example, the resulting DAG would contain `Int` twice as a parameter instead of actual values. In most cases, the value is not constant in either way and would contain `Int` as well.

Factory methods. Factory methods are covered in the paper but are not described how to detect them. As factory methods are a mere design pattern, there is no accurate and 100% reliable way to detect them. The only way is to decide them based on the naming or some heuristic methods. The implementation currently expects a factory method to be an invocation where the return type is the same as the owner type. This way of detecting works for the examples in the paper but not for cases where a separate factory class is used. When expanding upon the implementation of the code abstraction, a better way for handling factory methods would need to be developed.

Assignments on fields. Another case that is not documented are assignments on objects like, for example, `object.field = 15`. The question is whether this field assignment is handled as a method invocation. Most likely, all those assignments and similar operations, like `+=`, `-=` and so on, can be viewed in a functional programming manner, and therefore as "conceptual" method invocations. This way, the DAG can be constructed from them just as they would be from a regular method invocation, and the contextual information would be preserved.

3.2 Changes Pairing

The change pairing approach described in the paper works well but is very slow and can get very memory intensive. As mentioned before, matching the DAGs of two versions is done by comparing each DAG with every other. Matching like this is ok for a small number of DAGs, but as soon as either the file gets too big or the number of accepted classes (in the case of the paper, only seven crypto classes are taken) gets too big, the matching becomes very slow. For n DAGs, the matching is done in $O(n^2)$. Taking a file with 100 fields and 40 GB available RAM, the matching terminates after 4.5 minutes with an out-of-memory exception. The memory requirements are an issue regarding the references between the AST and the DAGs of the current implementation. They could be improved by removing the references or using a different parser. Nevertheless, nearly 5 minutes and ongoing for a single file is not applicable for a large number of files with a wide range of accepted classes.

Filtering the usage changes based on the removal/addition paths works well. The described method can easily be implemented, and the results reproduced. Refactorings, API introductions, and removals are eliminated effectively by the filtering mechanism, and only real usage changes remain. Only the duplication detection can be slow in cases where the number of remaining changes is significant, as every change is compared with every other remaining change, resulting in a runtime complexity of $O(n^2)$.

3.3 Usage Changes Clustering

Clustering the usage changes is described very briefly in the paper. A distance measure (Levenshtein similarity ratio [PZT+18]) is used in the paper to determine the similarity of two changes and cluster the changes. The exact properties used to calculate the ratio are omitted in the paper. For ease of implementation, for calculating the Levenshtein distance [AB09], the re-implementation takes the labels of the nodes. Taking the labels is probably not the best way to do it, but it is the closest to what is described in the paper.

Comparing all changes with each other is also a prolonged process. For n changes, the matching is done in $O(n^3)$. Therefore, using it on a large scale would most likely not be the best way to do it.

Doing actual clustering requires data. Unfortunately, finding data that matches all the criteria is not easy. Many of the repositories used for the paper are not available anymore or discontinued. Furthermore, the crypto API has changed since the paper was released, and many of the method signatures or classes are different. When checking the repositories used in the paper, only 50 changes out of 25,000 diffs had any usages of the crypto classes. However, none of those 50 changes had any instances of changes that were not filtered out. Analyzing those 25,000 diffs took about 15 minutes, noting that all changes were processed in sequence. Most of the change processing was canceled after building the DAGs due to not having any usages of the crypto classes. Taking all projects that are still available gives a total of 140 projects with a total of 170k files with ten versions each. Unfortunately, analyzing those 1.7 million changes can not be done due to the previously mentioned memory issues. Furthermore, it is not very computing-power efficient to analyze that many changes and discard almost all of them because they do not contain a change regarding the target API. The results that were shown in the paper and are available under http://diffcode.ethz.ch show that only 72k out of those 1.7 million changes even contain crypto API usages. Out of those 72k changes, only 186 were left after the filtering step. It is impressive what can be done with that little data, but having more data at hand could probably yield even better insights.

4 Conclusion

The paper presents a sophisticated approach for a new method for code analysis with the help of code changes. Unfortunately, the presented results are only suitable for specific niches, as, for example, shown in the paper, for crypto APIs. More specifically, the kind of analysis only works if the changes one is interested in are small and on a minimal scale. Expanding the analysis to a broader range or a larger volume of changes is not feasible, as the runtime would be too big.

Nevertheless, there are many possible directions one can go from this base approach. Following, three possibilities:

4.1 Enhance Code Change Detection

The code change detection with the DAG creation approach for each object worked well to detect an API's usages. The approach has too little context information for detecting changes in general and does not cover any control flow logic. For making it useful on a large scale, the generated DAGs need to be extended. It would probably make sense to generate a combination of the base AST and the DAGs from the paper. An example of such a combination for the code snipped (Listing 1) is shown in Figure 4. The AST would contain the control flow of the code and link to the DAGs for each object. There could exist multiple DAGs for the same object on different branches. This is visualized in the blue box.

```
1  class CryptoUtils {
2    private String password;
3
4    public CryptoUtils(String password) {
5      this.password = password;
6    }
7
8    private Cipher getCipherInstance(CipherMode mode,
         String key) {
9      Cipher ciph;
10     ciph = Cipher.getInstance("AES");
11
12     if (mode == CipherMode.ENCRYPT) {
13       ciph.init(Cipher.ENCRYPT_MODE, new
             SecretKeySpec(key.getBytes()));
14     } else {
15       ciph.init(Cipher.DECRYPT_MODE, new
             SecretKeySpec(key.getBytes()));
16     }
17
18     return ciph;
19   }
20
21   public String encrypt(String text) {
22     var cipher = getCipherInstance(CipherMode.
           ENCRYPT, password);
23     return cipher.encrypt(text);
24   }
25
26   public String decrypt() {
27     var cipher = getCipherInstance(CipherMode.
           DECRYPT_MODE, password);
28     return cipher.decrypt(text);
29   }
30
31 }
```

Listing 1. Basis for Figure 4

The two branches take a snapshot of the object before the branching point and extend it from there (visualized in red and green). Tracking the different states could be handled in a context object that gets passed to each tree and extends the parent's context. This context would then contain all other fields and variables with their respective values/reference to the DAG, thereby containing the code's data flow.

Creating hashes over the branches would give the ability to identify changes quickly. The identification of the objects should also be made in two levels. The first level is the object's identifier, and the second is the object's type. This identification provides the basis to differentiate between "real" changes or refactorings.

Having better code change detection would also offer a better comparison of versions like seen in, for example, Git[1].

4.2 End-to-end Toolchain

The paper "Deep Learning to Find Bugs [PSD17]" describes an end-to-end toolchain for taking code, generating buggy versions out of it, and then training a neural network with it to identify that buggy code. The main drawback of this

[1] https://git-scm.com/

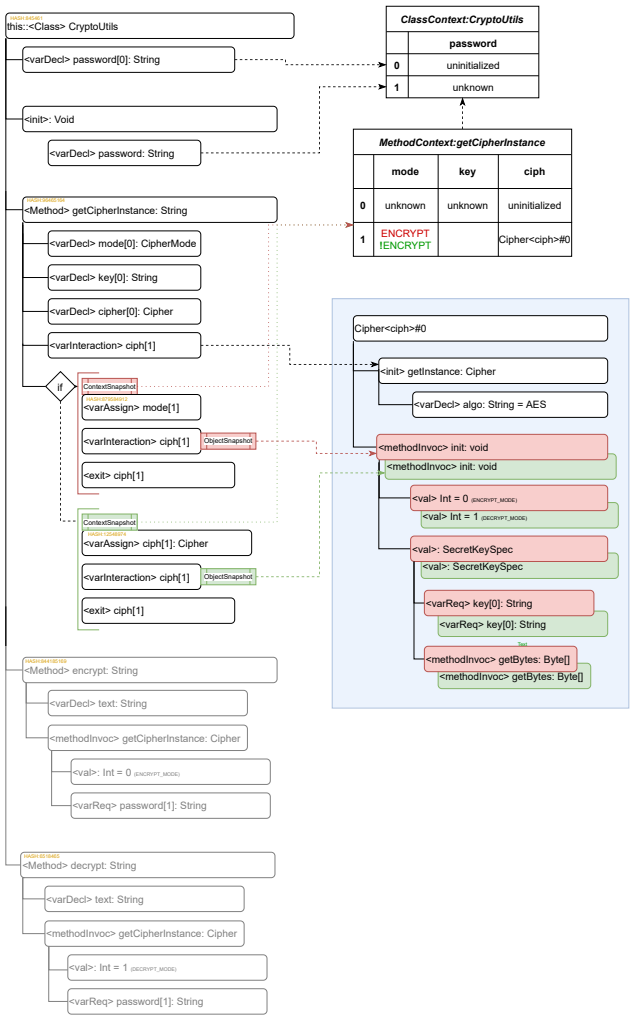

Figure 4. Combination of AST and DAGs, including its context information and hash values.

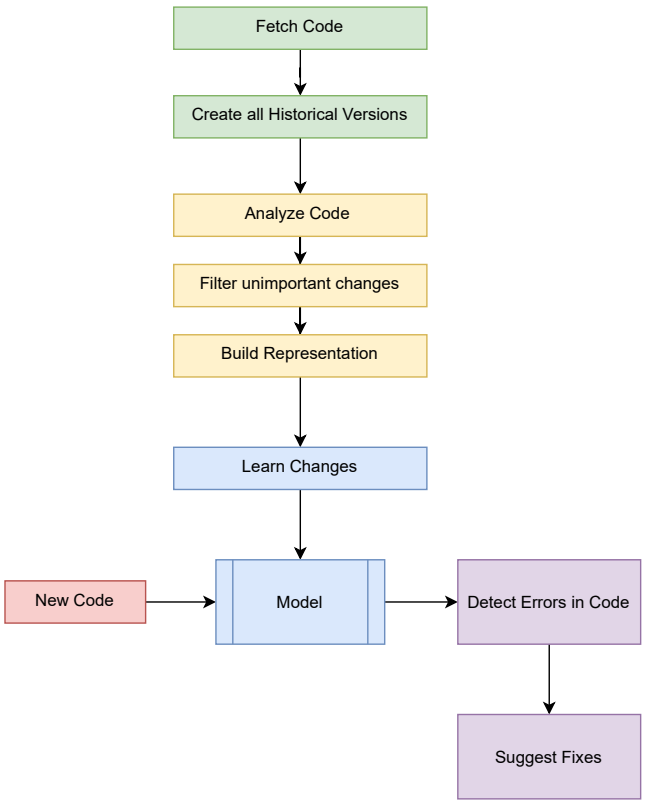

Figure 5. End-to-end toolchain for detecting code changes and suggesting fixes visualized

approach is that the bad code needs to be generated artificially. Projects with a version control system contain many corrections one could learn from, and there are also enough open-source projects available. With good code change detection that filters out bug fixes with high certainty, code changes can be used to train a neural network. Leveraging code changes would reduce the manual work that has to be done, learn on actual bugs, and even suggest the correct version in a second phase.

Figure 5 shows a possible toolchain. First code samples need to be farmed (marked in green). This is done by leveraging the power of big code, meaning publicly available code on GitHub, GitLab, Bitbucket, and co. Then the code versions need to be extracted by stepping through the code history and storing each version as a separate file. With this process done, the training data is set. Next, the code needs to be analyzed, and uninteresting or unimportant changes need to be filtered out (marked in yellow). The left changes can then be put into a representation that can be persisted. With that done, the creation and training of the model can be done (marked in blue). Finally the model can be used to detect errors for new code (marked in red) and possibly suggest fixes (marked in violet).

Even though this approach seems straightforward, it has some obstacles to overcome. Namely, smartly filtering out uninteresting code changes as described in subsection 4.1, and creating a good representation of the code that the network can learn. Furthermore, the code analysis process needs to be relatively fast to process all the data in a reasonable time for the training phase and for being a practical tool one wants to use.

4.3 Currently-Changing Dashboard

Having a dashboard for getting insights into your project's code has become more and more state of the art and is used in many projects. They give you an overview of your code quality and mark the hot spots, such as security issues, common bugs, or anti-patterns. Many of those insight tools, like,

for example, SonarQube[2] or the ETHZ spin-off DeepCode[3], now acquired by Snyk[4], resulted among others from the paper this one aims to reproduce, rely on more or less static rules to find errors in your code.

By having a solid change detection system (similar to the one in subsection 4.1), a dashboard could be developed that uses the changes of the open-source community (*Big Code*) to mark parts of a project that might need to be investigated. The system would not necessarily say what needs to change exactly but would indicate where to look. A change hotspot might be a common library that changed its API or a new language version with updated semantics. Furthermore, the system could show connections between projects and show wherein the code API upgrades, vulnerability and security issue fixes, and other changes are happening. A system like that could show trends and give the community a better feel for what is currently happening and where their focus should be.

References

[AB09] Mikhail J Atallah and Marina Blanton. Algorithms and Theory of Computation Handbook, Second Edition: General Concepts and Techniques. 2009.

[PSD17] Michael Pradel, Koushik Sen, and T U Darmstadt. Deep learning to find bugs, 2017.

[PZT+18] Rumen Paletov, Eth Zurich, Petar Tsankov ETH Zurich, Veselin Raychev, Martin Vechev ETH Zurich, Petar Tsankov, and Martin Vechev. Inferring Crypto API Rules from Code Changes. 2018.

[2] https://sonarqube.org
[3] https://www.deepcode.ai
[4] https://snyk.io