Smart Contract Similarity Detection

SE6387.001 Advanced Software Engineering Project

Team Members

Junaid Hashmi
Miao Miao
Jeongwon Seo

Table of Contents

I. Introduction **2**
- A. Problem 2
- B. Goals 2

II. Background **3**
- A. Solc Compiler 3
- B. Radare2 3
- C. Doc_2_Vec Model 3
- D. Asm_2_Vec Model 4
- E. Project Process utilizing the two Models 5

III. Use Cases **6**
- A. Proposed Use Cases 6
- B. Distribution of the Use Cases 6

IV. Architecture & Design **8**
- A. Overall Project Structure 8
- B. Internal Design of the Project 9

IV. Appendix **12**

List of Figures **14**

Works Cited **14**

I. Introduction

A. Problem

In recent years, the number of smart contracts running in the blockchain have increased dramatically, leading to growing concerns for security and vulnerability propagation caused by code reuse that can be deployed with malicious intents. Consequently, the effort for coming up with ways to detect similarity between smart contracts have also seen an increase. Our project team tried to address the above concern by attempting to use Doc2Vec and ASM2Vec approaches for smart contract similarity detection.

B. Goals

The goal of this project is to accomplish the following three things:

1. Given two smart contracts, automatically calculate their similarity using cosine similarity
2. Generate probability density function graphs to determine the threshold of the similarity score
3. Detect the co-cloned pairs and the non-co-cloned pairs of contracts along with their similarity values using the computed threshold

Below is a flowchart diagram that represents the overall goal of the project. The goal of this project is to design a system that can compare the similarity between two smart contracts by using the vectorization approach. Moreover, the system will be able to produce the Probability Density Graph from similarity values in order to determine the threshold. And then use the threshold to detect the co-clone pair of contracts from the contract repository.

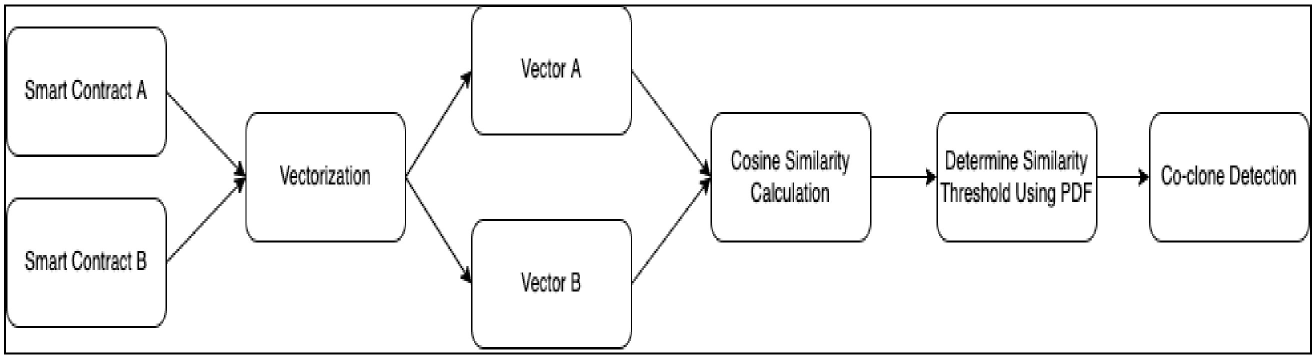


Figure 1-1. A flowchart diagram of the overall project goal

II. Background

A. Solc Compiler

Solc, Solidity Compiler, is a compiler that converts from a high-level solidity language into Ethereum Virtual Machine (EVM) bytecode so that it can be executed on the blockchain by EVM. It is a contract-oriented, high-level language whose syntax is similar to that of JavaScript ("Solidity").

B. Radare2

Radare2 (also known as r2) is an open-source software reverse-engineering framework to help disassemble, debug, analyze, compare data, and manipulate binary files. This framework helps convert the binary code to opcode that can be used for model training in language processing engines like word2vec, doc2vec, and asm2vec. In addition, r2 is an extremely useful reverse-engineering tool for analysis of suspect codes for plagiarism, software virus threats, and the like (Baldawa, "RADARE2 For Reverse Engineering-Part1").

C. Doc_2_Vec Model

Doc_2_Vec is used to create a numeric representation of a document, regardless of its length. It creates a vectorized representation of a group of words taken collectively as a single unit and doesn't only give the simple average of the words in the sentence. This model is implemented using the `gensim` library ("Gensim- doc2vec Model").

The procedure to generate the document vectors is:

Step 1: Train the model by creating the tagged document
Step 2: Initialize the model using the command gensim.models.doc2vec.Doc2Vec()
Step 3: Build the vocabulary
Step 4: Train the Doc2Vec Model
Step 5: Analyze the output by using model.infer_vector()

By following the steps above, an example output of the document vector would look something like the following:

```
[
   -0.2556166 0.4829361 0.17081228 0.10879577 0.12525807 0.10077011
   -0.21383236 0.19294572 0.11864349 -0.03227958 -0.02207291 -0.7108424
   0.07165232 0.24221905 -0.2924459 -0.03543589 0.21840079 -0.1274817
   0.05455418 -0.28968817 -0.29146606 0.32885507 0.14689675 -0.06913587
   -0.35173815 0.09340707 -0.3803535 -0.04030455 -0.10004586 0.22192696
   0.2384828 -0.29779273 0.19236489 -0.25727913 0.09140676 0.01265439
   0.08077634 -0.06902497 -0.07175519 -0.22583418 -0.21653089 0.00347822
   -0.34096122 -0.06176808 0.22885063 -0.37295452 -0.08222228 -0.03148199
   -0.06487323 0.11387568
]
```

D. Asm_2_Vec Model

Asm_2_Vec is an assembly clone search approach which learns a vector representation of an assembly function by discriminating it from the others (S. H. H. Ding, B. C. M. Fung, and P. Charland).

- It does not require any prior knowledge such as the correct mapping between assembly functions or the compiler optimization level used.
- It learns lexical semantic relationships of tokens appearing in assembly code and represents a function as an internally weighted mixture of latent semantics.

In Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland's study on this model titled "Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization", they mention an implementation approach that is used to train such a model. They first propose to model a control flow graph, which is a representation of an assembly function generated by radare2 (also known as r2), as multiple sequences. In a control flow graph, or a CFG, each sequence will correspond to a potential execution trace that will contain assembly instructions (S. H. H. Ding, B. C. M. Fung, and P. Charland)

Then, the procedure will be:
Step 1 & 2: Train a representational model and produce a numeric vector for each repository function

- Map a repository function to a vector which will be the vector representation of the function to be learned in the training model.
- The operands and operations in the opcode instructions will be considered as tokens.
- Each token will then be mapped into two different numeric vectors.

Step 3: After training, this will represent a token's lexical semantics.

- This will help visualize the relationship among the tokens (which can help with the prediction).

Although the Asm_2_Vec model is convincing, there are a few limitations. First, it is designed for a single assembly code language and the clone search engine is architecture-agnostic. Which means this is not directly applicable for semantic clones across architectures yet. Second, the current selective callee expansion mechanism cannot determine the dynamic jumps such as a jump table. Lastly, but most importantly, this is a black box static approach which means it cannot explain or justify the returned results by showing the cloned subgraphs or proving symbolic equivalence. In other words, it has limited interpretability.

E. Project Process utilizing the two Models

Below is a flowchart diagram that represents the workflow of the project. The solc compiler will first compile the smart contracts into binary code and opcode. Then radare2 can convert the binary code into assembly code to prepare the data for model training. The system will support two vectorization models: Doc_2_Vec and Asm_2_Vec. Once the model training is done, the system can generate the probability density graphs from the similarity values gained by the model and compute sim threshold by using these PDF graphs. Finally, the system will be able to detect the co-clone contract pairs based on the threshold.

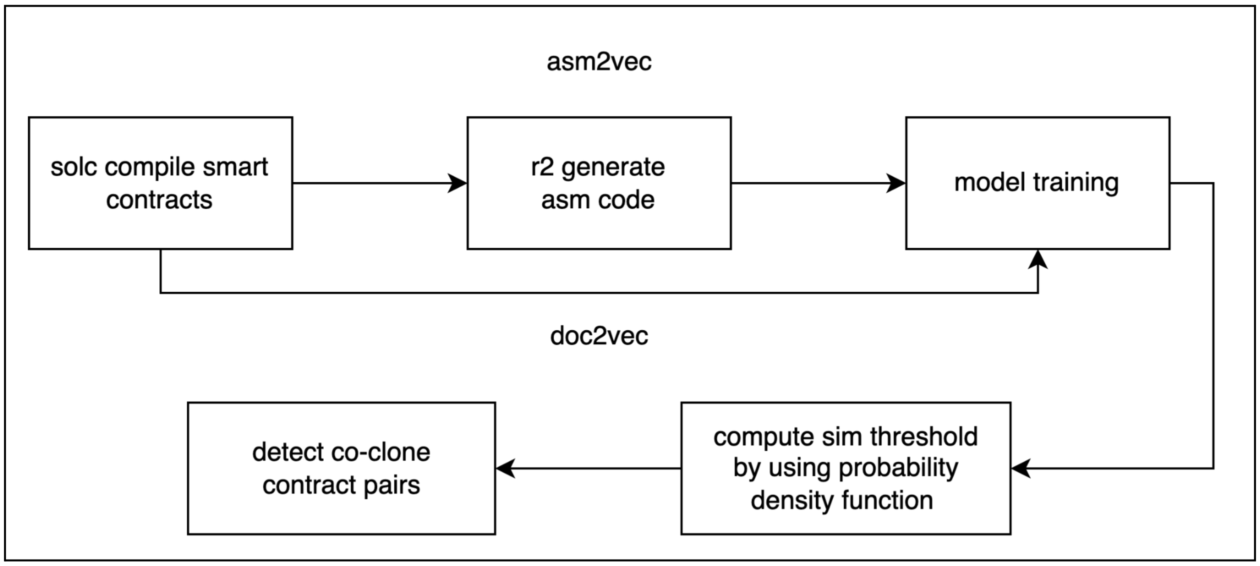

asm2vec
solc compile smart contracts
r2 generate asm code
model training
doc2vec
detect co-clone contract pairs
compute sim threshold by using probability density function

Figure 2-1. A flowchart diagram of the project process

III. Use Cases

A. Proposed Use Cases

The following is the list of proposed use cases:

- Preprocess Data
- Compile Source Code to Opcode & Binary Code
- Convert Binary Code to Assembly Code
- Train the model under Doc2Vec Model
- Train the model under Asm2Vec Model
- Generate the Probability Density Function Graphs
- Distinguish the Co-Cloned Contracts and the Non-Co-Cloned Contracts

B. Distribution of the Use Cases

The following is the list of use cases distributed into three iterations:

Iteration 1:

ID: UC1	**Title**: Preprocess Data	**Priority**: 1
Description: Filter out bad contracts based on the white list and black list. Split all smart contracts into 80% for training and 20% for testing.		
Preconditions: Smart contracts, black and white lists are accessible		
Postconditions: Py-sloc-x and radare2 libraries are correctly integrated		

ID: UC2	**Title**: Compile Source Code to Opcode & Binary Code	**Priority**: 1
Description: Given smart contract source code, using solc compiler to generate opcode and binary code to prepare data for Doc_2_Vec model training		
Preconditions: Smart contracts are filtered and splitted into training and testing sets		
Postconditions: Gensim and Asm_2_Vec libraries are correctly integrated		

ID: UC3	**Title**: Convert Binary Code to Assembly Code	**Priority**: 1
Description: After solc generates binary code, using radare2 to analyze control flow graph and convert binary code to assembly code to prepare data for model training.		
Preconditions: Binary code is correctly generated by solc and accessible by radare2.		
Postconditions: Gensim and Asm_2_Vec libraries are correctly integrated		

Iteration 2:

ID: UC4	**Title**: Train the model under DOC_2_VEC Mode	**Priority**: 1
Description: Using the functions provided by Gensim library to create, train, save a DOC_2_VEC model.		
Preconditions: Training set is preprocessed and accessible		
Postconditions: A DOC_2_VEC model is successfully trained and saved		

ID: UC5	**Title**: Train the model under ASM_2_VEC Mode	**Priority**: 1
Description: Using the functions provided by ASM_2_VEC library to create, train, save an ASM_2_VEC model.		
Preconditions: Training set is preprocessed and accessible		
Postconditions: An ASM_2_VEC model is successfully trained and saved		

Iteration 3:

ID: UC6	**Title**: Generate the Probability Density Function Graphs	**Priority**: 1
Description: Compute the probability density from a certain dataset to determine the threshold similarity value		
Preconditions: All similarity values are generated by the trained models and the data is accessible		
Postconditions: The probability density function graphs are generated as a PNG image		

ID: UC7	**Title**: Distinguish the Co-Cloned Contracts and Non-Co-Cloned	**Priority**: 1
Description: Using the threshold value from UC5, find the co-cloned contract pairs and non-co-cloned contract pairs		
Preconditions: A threshold value from the probability density graph is determined		
Postconditions: All relevant data is written to a newly generated CSV file and correctly exported		

IV. Architecture & Design

A. Overall Project Structure

Below is the diagram that shows the overall project structure.

The 'Smart-Contract-Similarity-Detector' is the root folder of the project. The 'External Library' folder contains the dependencies of the project. Inside the 'Smart-Contract-Similarity-Detector' folder, 'project_tests' and 'src' each contain the test cases and source code of the project. Inside 'src' folder, 'contracts' contains the smart contract source code, 'out' contains the opcode and binary code generated by solc, 'model' contains 2 vectorization models, 'result' contains the similarity reports and 'settings' contains the black and white list files.

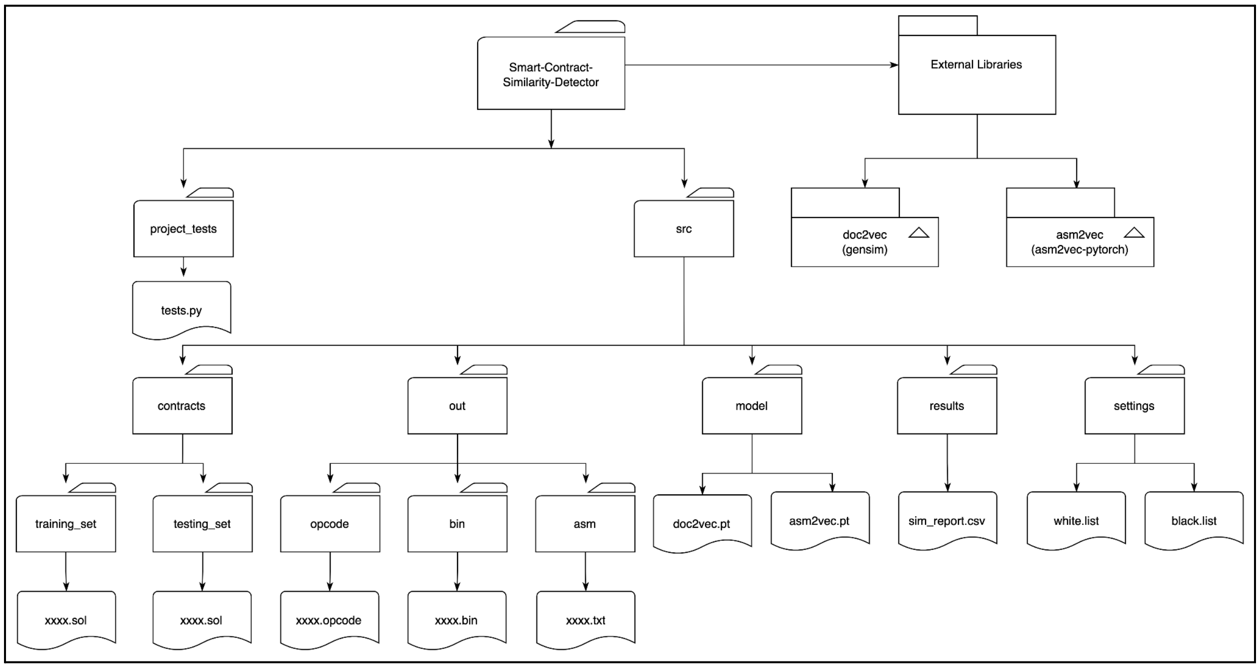


Figure 4-1. Diagram of the Overall Project Structure

B. Internal Design of the Project

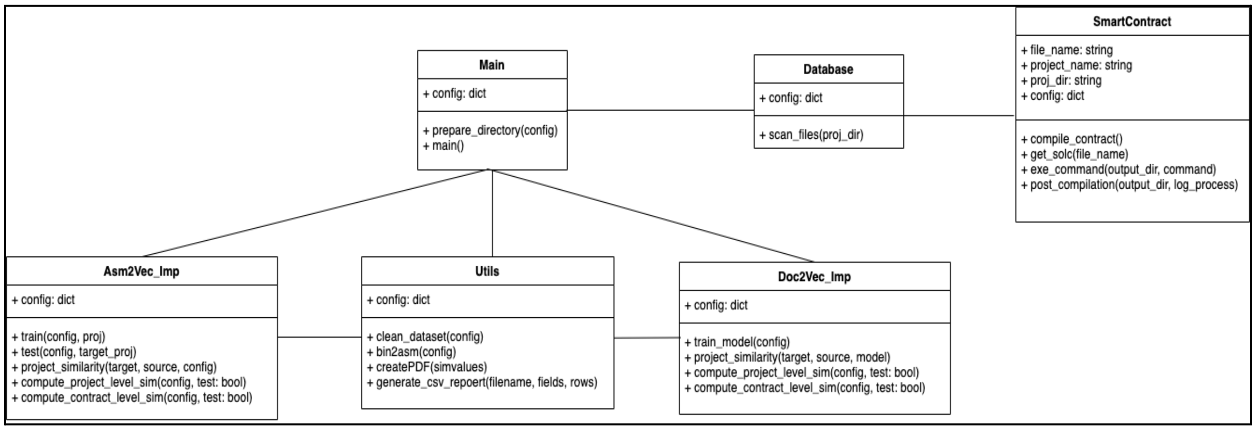

SmartContract
+ file_name: string
+ project_name: string
+ proj_dir: string
+ config: dict
+ compile_contract()
+ get_solc(file_name)
+ exe_command(output_dir, command)
+ post_compilation(output_dir, log_process)
Main
+ config: dict
+ prepare_directory(config)
+ main()
Database
+ config: dict
+ scan_files(proj_dir)
Asm2Vec_Imp
+ config: dict
+ train(config, proj)
+ test(config, target_proj)
+ project_similarity(target, source, config)
+ compute_project_level_sim(config, test: bool)
+ compute_contract_level_sim(config, test: bool)
Utils
+ config: dict
+ clean_dataset(config)
+ bin2asm(config)
+ createPDF(simvalues)
+ generate_csv_repoert(filename, fields, rows)
Doc2Vec_Imp
+ config: dict
+ train_model(config)
+ project_similarity(target, source, model)
+ compute_project_level_sim(config, test: bool)
+ compute_contract_level_sim(config, test: bool)

Figure 4-2. Class Diagram of Smart-Contract-Similarity-Detector Project

Below are the classes that are used in this project with brief descriptions:

- Main: Main class is responsible for controlling the main flow of the project, which includes data set cleaning, splitting training and testing sets, project compilation, model training, similarity computation, etc.

+ `prepare_directory(config)`: This function automatically creates the directories for the program to use. The list of directories that it creates is: out, result, training, testing, opcode, and asm (testing, testing_opt, training, training_opt). And this list of generated directories is demonstrated in Figure # as well.

- Database: Database class is responsible for scanning projects under a specified dataset directory.

+ `scan_files(proj_dir)`: This function is used to scan all files in the project directory and to creatine instance of the Smart Contract class.

- SmartContract: SmartContract class is responsible for compiling contracts under optimized and non-optimized mode, preparing data for model training.

+ `compile_contract`: This function is called to compile all of the projects.
+ `get_solc(file_name)`: This function is called to create a solc instance. Within this function, there are sub-methods that are called such as `get_pragma` and `remove_void`.
+ `exe_command(output_dir, command)`: This function is to execute the solc compiler commands to successfully compile the projects.
+ `post_compilation(output_dir, log_process)`: This function is to remove any empty files, format the opcodes, and to log the compilation messages on the console.

- **Asm2Vec_Imp**: Asm2Vec_Imp class is responsible for creating and training the Asm_2_Vec model, computing the threshold for co-clone pairs detection, detecting co-clone pairs and generating csv reports under Asm_2_Vec mode.
  + `train(config, proj)`: This function is used to train the model under the ASM_2_VEC model.
  + `test(config, target_proj)`: This function is to show the predicted probability results when testing.
  + `project_similarity(target, source, config)`: This function is to compute the actual similarity value between any two projects.
  + `compute_project_level_sim(config, test: bool)`: This function is to loop through the data in both testing and testing_optimized directory to compare the project-level similarity between each other.
  + `compute_contract_level_sim(config, test: bool)`: This function is to loop through data in both testing and testing_optimized directory to compare the contract-level similarity between each other.

- **Doc2Vec_Imp**: Doc_2_Vec_Imp class is responsible for creating and training the Doc_2_Vec model, computing the threshold for co-clone pairs detection, detect co-clone pairs and generate csv reports under Doc_2_Vec mode.
  + `train(config, proj)`: This function is used to train the model under the DOC_2_VEC model.
  + `project_similarity(target, source, config)`: This function is to compute the actual similarity value between any two projects.
  + `compute_project_level_sim(config, test: bool)`: This function is to loop through the data in both testing and testing_optimized directory to compare the project-level similarity between each other.
  + `compute_contract_level_sim(config, test: bool)`: This function is to loop through data in both testing and testing_optimized directory to compare the contract-level similarity between each other.

- **Utils**: Utils class is responsible for providing utility methods for other classes to use, such as converting binary code to assembly code, filtering out bad projects, splitting training and testing data sets, creating PDF graph and generating CSV files.
  + `clean_dataset(config)`: This function is to filter the bad, good, and the unknown projects using the white and the black lists.
  + `bin2asm(config)`: This function is for radare (r2) to convert the binary code to asm.
  + `createPDF(simvalues)`: This function is used to create the probability density function graphs to determine the similarity value threshold. This threshold value is used to distinguish between the co-cloned contract pairs and non-co-cloned contract pairs.

+ `generate_csv_report(filename, fields, rows)`: This function is used to export all generated data into a csv file.

IV. Appendix

This appendix outlines the procedure to be followed by a user of this tool for similarity detection between a target smart-contract against any number of source smart-contracts.

The setup required falls into three categories:

- Environment Prerequisites:
  Python3.8 or higher
  Python packages required: os, pathlib, typing, shutil, csv, subprocess, random, math, json, torch, tqdm, numpy, matplotlib, r2pipe, gensim, and solcx
  Solicity compiler: https://github.com/ethereum/solidity
  Radare2 decompiler: https://github.com/radareorg/radare2

- Tool configuration:
  Tool can be obtained from the public git repository by:
  ```
  git clone https://github.com/mileyseo/Smart-Contract-Similarity-Detector
  ```
  This would result in the project tree being cloned at your machine in the current directory.

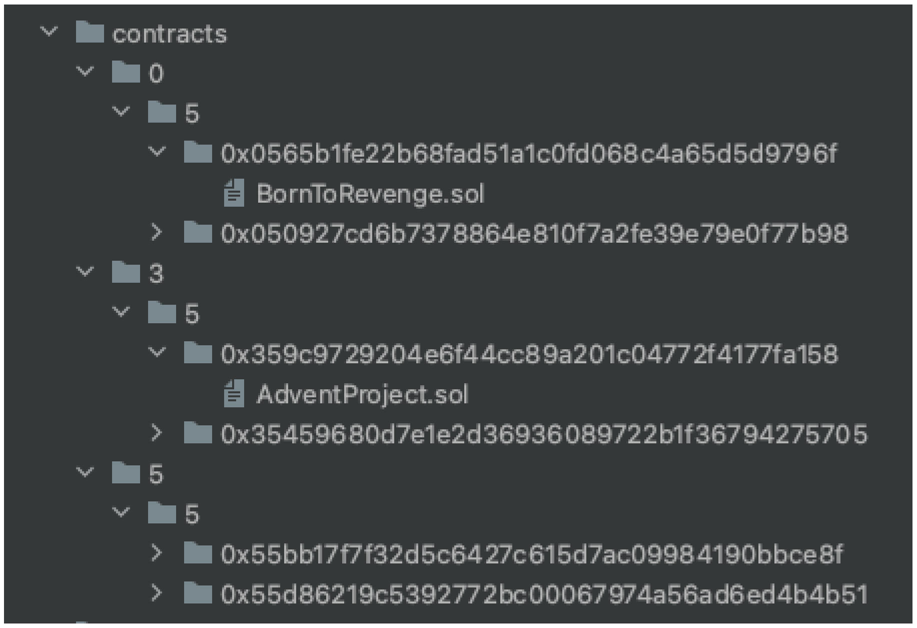
  The only file that may need to be edited is "config.json" that is found inside the "src" sub-folder. All the path definitions in this file are relative, so they can be used with their default values. The parameter "ASM_2_VEC" can be set to "true" or "false". By default, it is set to "false", which means the tool will use 'word2vec' model (gensim) for training and testing. If it is set to true, asm2vec engine will be used instead.

  If asm2vec is used, the current parameters under "ASM_CONFIG" should be ok in most cases. Please consult asm2vec documentation if further tuning is desired.

  For either doc2vec or asm2vec, there are "THRESHOLD" sets for identifying a 'clone'. These values were obtained as a result of analyzing pdf graphs of similarity comparisons against 5000 to 7000 thousand smart contracts. These thresholds should serve well in most cases, unless one wishes to retrain the models using a much larger number of contracts.

  The tool can be used with existing models, or can be trained using smart contracts one can obtain from a number of online repositories.

- File structure and setup:
  1. Create a ```/contracts``` directory under ```/src/```
  2. Go to this github repository https://github.com/PrCatch/etherscan-0815-0914
  3. Download project as a ZIP file in another folder
  4. Extract the project from the ZIP file
  5. Select the contracts you want to compare from the ```/contracts``` folder you unzipped
  6. Place those contracts into ```/src/contracts``` directory
     * The contracts should be placed following the format of ```/#/#/project```
     * Example:



```
contracts
  0
    5
      0x0565b1fe22b68fad51a1c0fd068c4a65d5d9796f
        BornToRevenge.sol
      0x050927cd6b7378864e810f7a2fe39e79e0f77b98
  3
    5
      0x359c9729204e6f44cc89a201c04772f4177fa158
        AdventProject.sol
      0x35459680d7e1e2d36936089722b1f36794275705
  5
    5
      0x55bb17f7f32d5c6427c615d7ac09984190bbce8f
      0x55d86219c5392772bc00067974a56ad6ed4b4b51
```

List of Figures

1. Figure 1-1. A flowchart diagram of the overall project goal
2. Figure 2-1. A flowchart diagram of the project process
3. Figure 4-1. Diagram of the Overall Project Structure
4. Figure 4-2. Class Diagram of Smart-Contract-Similarity-Detector Project

Works Cited

Baldawa, Sandeep. "RADARE2 For Reverse Engineering-PART1." Medium, ITNEXT, 23 Sept. 2020, https://itnext.io/radare2-for-reverse-engineering-part1-eedf0a47b5cc.

"Gensim - doc2vec Model." Tutorials Point, https://www.tutorialspoint.com/gensim/gensim_doc2vec_model.htm.

S. H. H. Ding, B. C. M. Fung and P. Charland, "Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization," 2019 IEEE Symposium on Security and Privacy (SP), 2019, pp. 472-489, doi: 10.1109/SP.2019.00003.

"Solidity." Solidity, https://docs.soliditylang.org/en/v0.8.17/.