Reverse-Engineering Database – An IDA-Pro Plug-in

Testing the project

## 1. Introduction

- The purpose of the server is to supply the user with Public Descriptions fitting his functions.
- Attributes characterizing the function are gathered in the client-side, and sent to the server by the user.
- Using the attributes, the server looks in its DB for functions similar to the user's function.
- In order to do that, the server computes a similarity grades (or a matching grades) between the user's function and the functions it has stored. It does that using several Heuristics.
- A heuristic, in the context of this project, is a way of comparing two functions, using their attributes, and quantifying their similarity.
- The server computes the similarity grade based on several heuristics using several attributes. Each heuristic-attribute pair has its own importance, and thus is given a different weight in the final grade (which is simply a weighted average).
- The server returns a description to the user, only if the function the description is connected to, has high similarity with the user's function, i.e. the similarity grade is higher than a certain threshold.
- **The following questions rise:**
    - **What weight should be given to each heuristic-attribute pair when computing the final similarity grade?**
    - **How best to determine the optimal threshold?**
    - **What is "optimal" in the context of this project?**
- With the data gathered in the course of testing the project, we can try and answer additional questions, such as:
    - **What is the value of the similarity grade when based on <u>one</u> heuristic-attribute pair only (when none of the others is given any weight)?**
    - **How does each of heuristic-attribute pairs influence the final similarity grade?**
    - **By which heuristic-attribute pair can we induce two functions are similar?**

## 2. The Tests

We would like to determine one set of weights is better than another. "Better" needs a definition. **In our basic test we look at a pair of functions which have similar functionality. We check whether the similarity grade they received is higher than the threshold we set.** We would like to receive low similarity grades for functions which are of different functionality, and higher similarity grades for functions that are actually similar.

We need an objective way of telling whether two functions are of similar functionalities. We can't use the tools we developed to determine this, as it would be biased. We also can't mass reverse-engineer functions in order to find similar and non-similar functions, as it would be inefficient.

What we chose to do is look at a pair of executable files which are different versions of the same application. The executable files will be the results of compilations of the source codes for debugging purposes (as opposed to release purposes).

This way the symbols (function names from the source codes) will be available to IDA. Some functions will be modified from one version to another though the names will probably remain the same.

For the purpose of testing, we say two functions have similar functionality if they have the same name.

Given two executable, we choose one of the executable files arbitrarily and refer to it as "the first executable", the other one will be referred to as "the second executable".

We handle only with functions whose names are in the intersection of the function names of the two files. We compare each of the handled functions in the first executable with all of the handled functions in the second executable. We do the same with each of the second executable's functions.

We get a two-dimensional array of similarity grades. If the functions on both axes are sorted in the same order (by their names, in alphabetic order for example), we expect the diagonal to contain high grades, and grades which are not on the diagonal to be low.

## 3. The Measures

We define two measures by which we examine a set of tests:

1. **Minimum of the False positive, False negative Ratios**
   We define a **false-positive** as the case when a high similarity grade (above threshold) is computed for a test even though the two functions compared have different names.
   Similarly, we define a **false-negative** as the case when a low similarity grade (below threshold) is computed for a test even though the two functions compared have the same name.
   Given a **set of test results** (a two-dimensional array of test results, each between a pair of functions) and a **threshold**, the **false-positive ratio** equals the number of false-positive results divided by the number of pairs with high similarity grade (above the given threshold).
   Similarly, the **false-negative ratio** equals the number of false-negative results divided by the number of pairs with low similarity grade (below the given threshold).

In this measure, given a set of test results, we find a threshold such that the maximum between the false-positive ratio and the false negative ratio is minimal (over all thresholds checked). **And return that maximum number**. (We find that threshold by iterating over grades from 0.0-1.0).

2. **Averages Difference**
   In this measure, given a set of test results, we average:
   a. The grades on the diagonal we mentioned before.
   b. The rest of the grades.
   We reduce the second from the first and return the results.

**One set of results is better than the other if the first measure is smaller for the first than for the second, i.e. the *"Minimum of the False positive, False negative Ratios"* for the first is smaller than *"Minimum of the False positive, False negative Ratios"* for the second.**

**We use this measure to determine optimal weights, because each set of weights results in a different array of grade.**

4. Finding the Optimal Weights

We'll concentrate on the following heuristic-attribute pairs:

1. ins_data_list_attr - an ordered list of integers representing the instructions themselves.
2. lib_calls_list_attr - an ordered list containing the library function names for library calls which occur in a function.
3. graph_rep_attr - a representation of the function's control-flow.
4. str_list_attr - an ordered list of the strings which appear in the function.
5. ins_type_attr - an ordered list of instruction types.
6. imm_list_attr - an ordered list of immediate values.

These are the pairs we currently use for computing the similarity grade.

**The process of finding the optimal weight is as follows:**

1. Choose two versions of the same executable.
2. For each permutation of weights:
   a. Compare functions from the first executable with functions in the second executable: pairs of compared functions will be comprised of one function from the first executable and one from the second. Functions compared are those whose names appear in the intersection of names from both executables. The result would be a two dimensional array of similarity grades.
   b. Compute the *"Minimum of the False positive, False negative Ratio"* for this two dimensional array.

c. If this result is smaller than the ones received for the weights checked so far, keep the weights.

3. Return the minimal weights.

- Each weight is in the scale of one to ten. In order to lower the number of sets (permutations) of weights we:
    - Dropped permutations where the weights had a common divider bigger than one.
    - Ran the algorithm on smaller scales first, so as to find the "order of importance", and then rearranged the heuristic-attribute pairs by this order (the most important first). We then chose only monotonically non-increasing permutations.

  These two changes lowered the number of permutations and the time it took to iterate them drastically.

<u>The optimal weights</u>

This process was repeated for several pairs of versions of an application called "tcping", which is used for pinging over a TCP connection. The number of function in the intersection is 97 functions. So, about 9404 comparisons were made at each test type.

The heuristic-attribute pairs, in the order in which their grades appear in the table:

ins_data_list, lib_calls_list, graph_rep, str_list, ins_type_list, imm_list

| Version | 0.2 | 0.4 | 0.6 | 0.8 |
|---|---|---|---|---|
| 0.4 | **Ratio**: 0.025 **Weights**: (2,8,10,2,10,1) | - | - | - |
| 0.6 | **Ratio**: 0.027 **Weights**: (2,9,9,2,10,1) | **Ratio**: 0.026 **Weights**: (2,2,10,2,10,1) | - | - |
| 0.8 | **Ratio**: 0.027 **Weights**: (2,9,9,2,10,1) | **Ratio**: 0.027 **Weights**: (2,9,9,2,10,1) | **Ratio**: 0.026 **Weights**: (2,9,9,2,10,1) | - |
| 0.10 | **Ratio**: 0.027 **Weights**: (2,9,9,2,10,1) | **Ratio**: 0.027 **Weights**: (2,9,9,2,10,1) | **Ratio**: 0.026 **Weights**: (2,9,9,2,10,1) | **Ratio**: 0.026 **Weights**: (3,3,3,3,3,1) |

- The ratios in the top table are the optimal ones. They were received for threshold 0.9 (we constantly received 0.9 is the optimal threshold) and the weight sets written in the table.
- We received an anomaly when comparing versions 0.8 and 0.10.

The following graph represents optimal weights received in two comparisons:

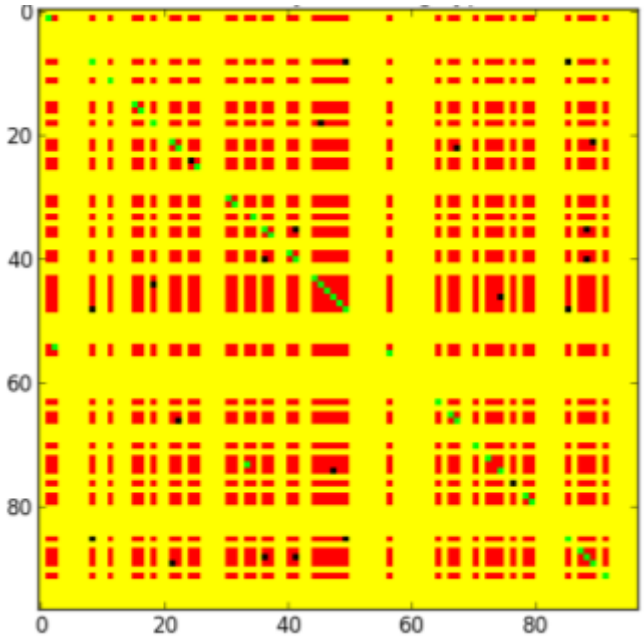

- Averaging the results, the following weights are received:
  - ins_type_list – 10
  - graph_rep – 9
  - lib_calls_list – 9
  - ins_data – 2
  - string_list – 2
  - imm_list – 1
- When we compared executable files from other applications, some pairs received different weights. For example, in the results above the string attribute receives a low weight, though when comparing other executables (e.g. Notpad++) it receives a higher weight. A possible explanation is that "tcping" has a small amount of strings, compared to the other applications.
- This implies that haven't found the optimal weights for all execuables, but only for "tcping". In addition, "tcping" falls under the network application category. It can be assumed that different sets of weights will characterize different application categories. For example, GUI applications will contain more strings than application which don't contain GUI (Malware for instance).
- Two heuristic-attribute pairs which generally get high weight are the ins_type_list and the graph_rep. This implies that by using only these two pair, you can show similarity in high probability.
- The fact that we received weights 1 and 10 implies there is room for repeating this test in a larger scale, though the question of what scale is large enough cannot be answered for the results. It probably depends on the level of preciseness we wish to achieve.

5. Presenting the results

We use two types of graphs to give the reader an intuition about the results. Both are generated using Matplotlib, a Python library for plotting graphs:

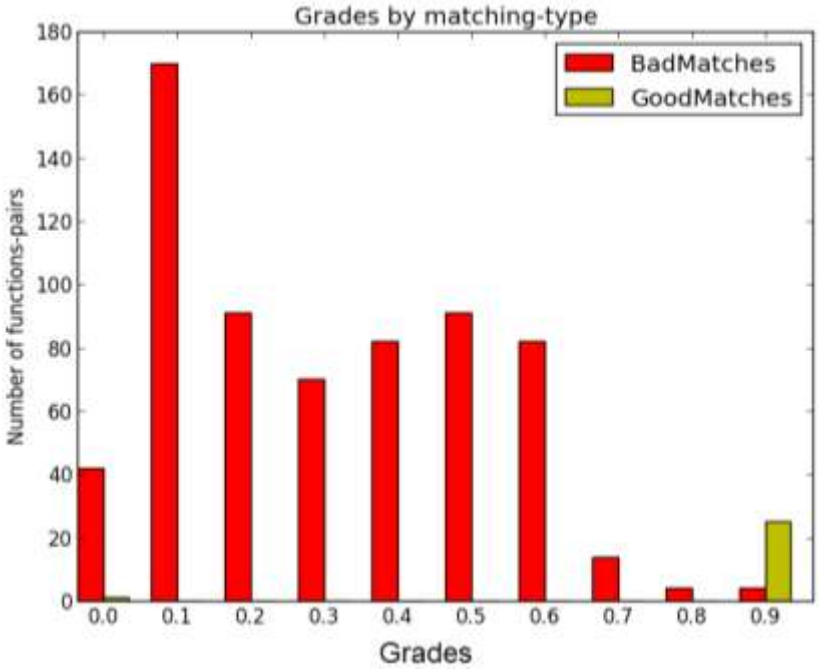
1. Success/Failure Graph – closely related to the two dimensional array of similarity grades discussed before. The pivot is located at the top left corner. Both axes represent functions from an executable file, each a different file. A function has a place on one axis if a function of the same name exists in the other executable. A pixel is colored by considering the function represented by the row the pixel is in, and the function represented by the column the pixel is in:
    a. If the two functions share the same name and their similarity grade is higher than the threshold, **the pixel is colored green**.
    b. If the two functions have a different name and have a similarity grade lower than the threshold, **the pixel is colored red**.
    c. If the two functions share the same name but their similarity grade is lower than the threshold, or the two functions have a different name but their similarity grade is higher than the threshold, **the pixel is colored black**.
    d. In some cases we systematically chose not to compare. For instance, when comparing two functions, which don't contain strings, by their string lists. In this case **the pixel is colored yellow**.

    In general, green and red are good and black is bad, yellow is of no importance.



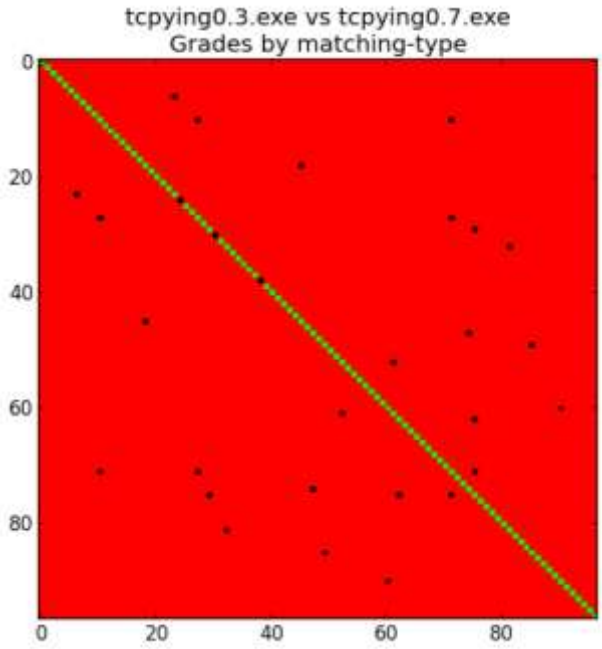

2. Grade Distribution – what grades were computed for a set of tests? Good matches refer to comparisons in which the two functions share the same name. Bad matches

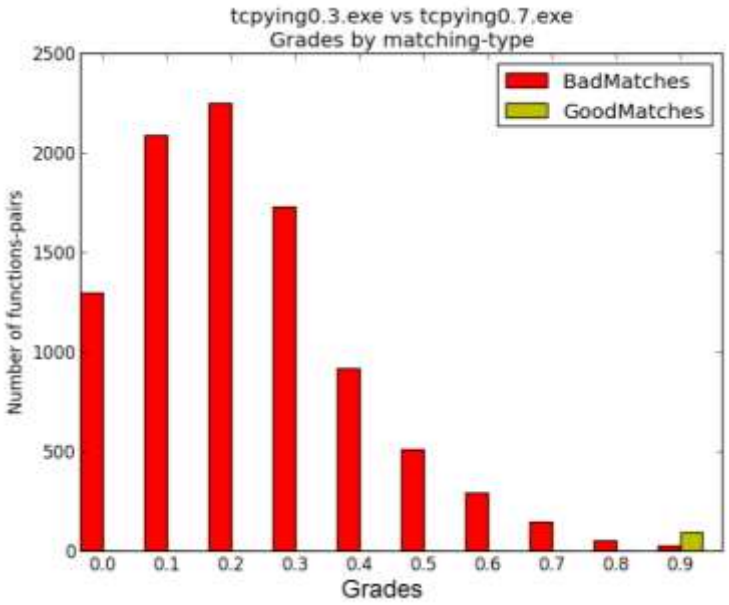
refer to comparisons in which the two functions have different names.



6. Testing the matching Grade

After setting the heuristic-attribute pairs' weights to be the ones we found optimal, we wish to check the preciseness of the similarity grade. We compare two versions of "tcping" that haven't been used to find the optimal weights.




- The optimal threshold is 0.9.
- The averages difference for the results is 0.9.
- The maximum of the false-positive, false-negative ratios is 0.026.

7. Testing the heuristic-attribute pairs

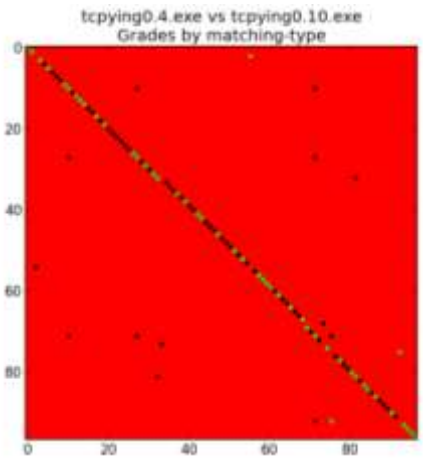
We wish to examine each heuristic-attribute pair by itself. We do this by repeating the tests we have done thus far for the matching grade, except give a weight to one of the pairs at each time (all the others get zeroes as weights).

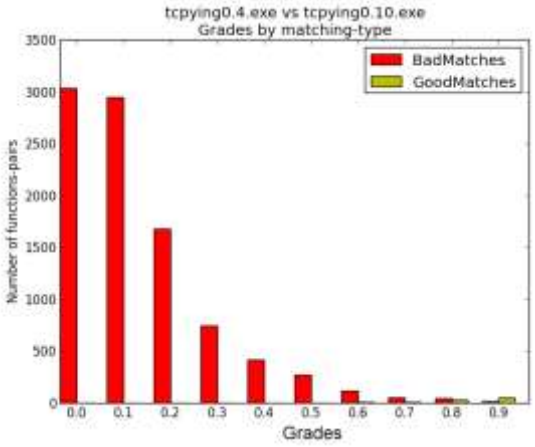
The results are shown below:

| Heuristic - Attribute | Measure | 0.2 vs 0.4 | 0.2 vs 0.10 |
|---|---|---|---|
| ins_data_list | Averages Difference | 0.674 | 0.665 |
| | Min FalsePos-FalseNeg Ratio | 0.060 | 0.071 |
| lib_calls_list | Averages Difference | 0.971 | 0.947 |
| | Min FalsePos-FalseNeg Ratio | 0.315 | 0.327 |
| graph_rep | Averages Difference | 0.612 | 0.632 |
| | Min FalsePos-FalseNeg Ratio | 0.137 | 0.137 |
| str_list | Averages Difference | 0.972 | 0.966 |
| | Min FalsePos-FalseNeg Ratio | 0.0* | 0.0* |
| ins_type_list | Averages Difference | 0.662 | 0.660 |
| | Min FalsePos-FalseNeg Ratio | 0.076 | 0.078 |
| imm_list | Averages Difference | 0.524 | 0.500 |
| | Min FalsePos-FalseNeg Ratio | 0.594 | 0.611 |

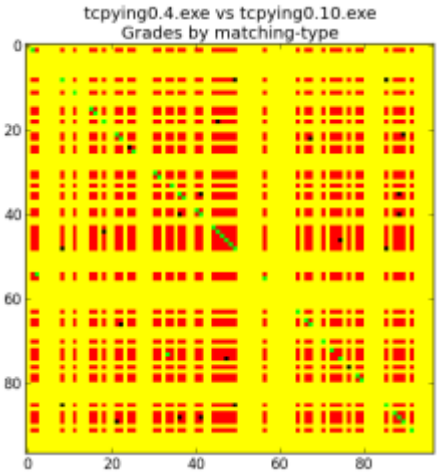
The matching graphs:

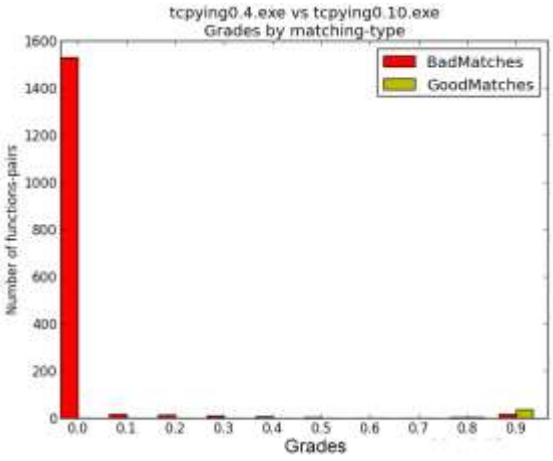
ins_data_list



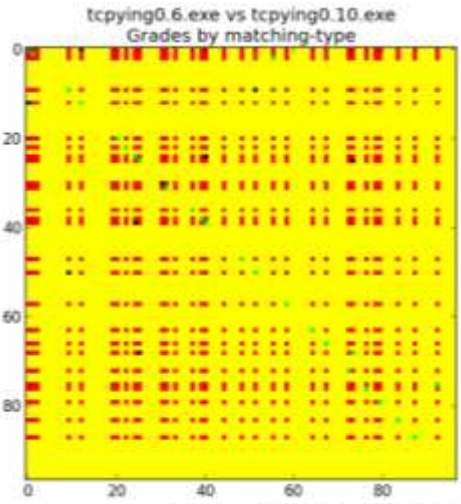



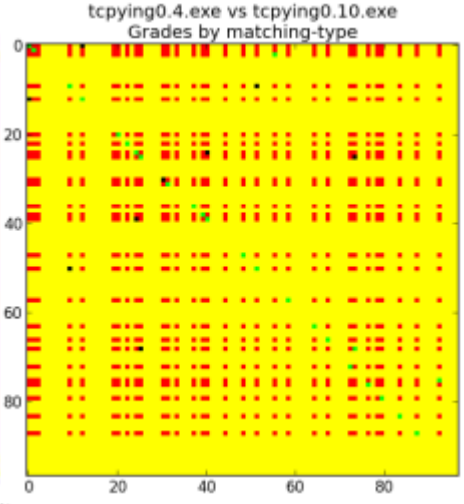
lib_calls_list

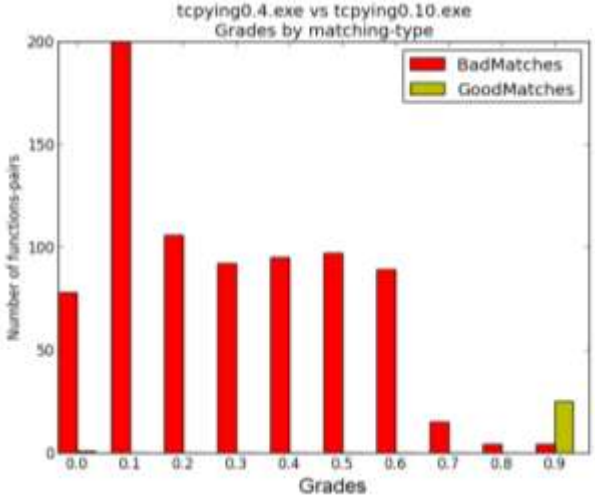


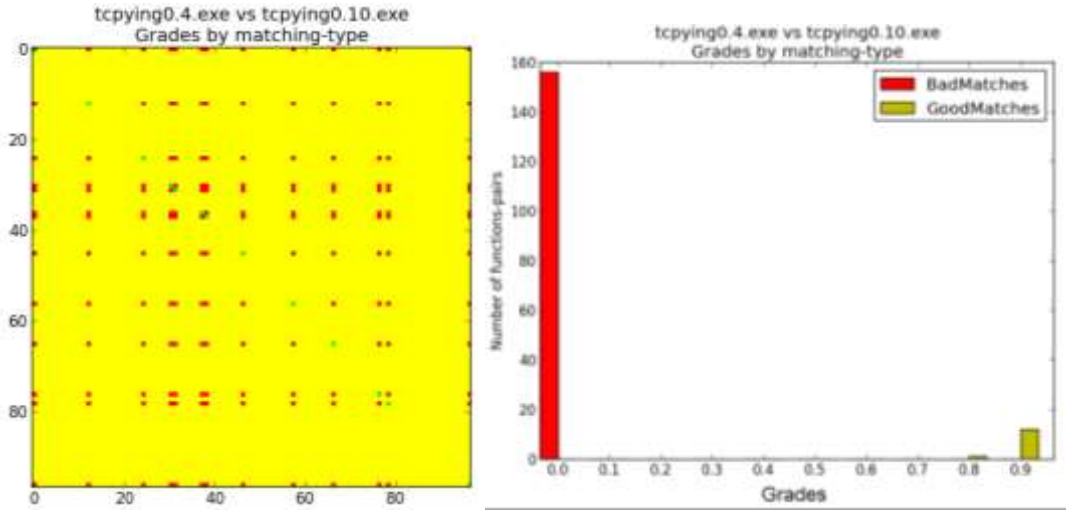

graph_rep



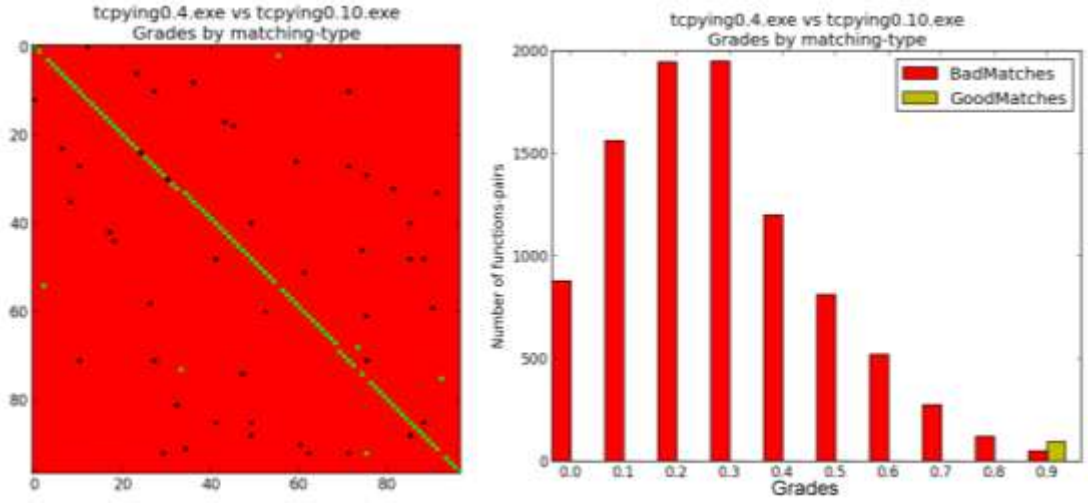
str_list




Note the almost-perfect separation between pairs of matching functions and pairs of non-matching functions. This is due to the fact that it is highly unlikely for two non-similar functions to have common strings.

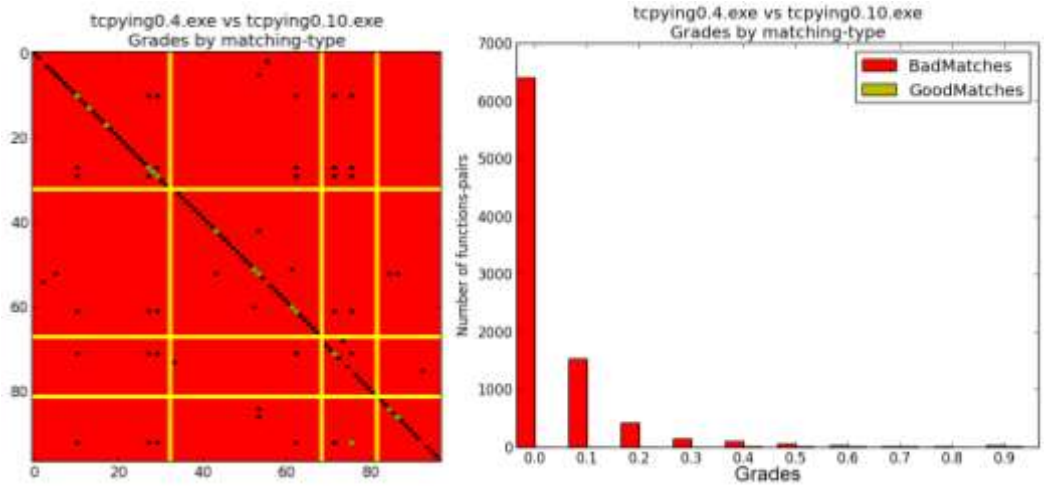
Nonetheless, observe the big amount of pairs we chose not to compare, due to lack of strings.

ins_type_list




As can be expected, there are no yellow pixels, since every function contains instructions and thus can be compared by their types.

imm_list




Notice the false-negatives – we do not find similarity for a lot of the similar function pairs (those pairs who share names). This is because at the time of conducting the experiment and generating the graph, the attribute was not good enough: when we extracted immediate values from the functions, we extracted constant instruction addresses as well as immediate values. As the addresses change from version to version, and their number is generally greater than the actual immediate values, we mistake similar pairs to be non-similar.

Removing Heuristic-Attribute pairs from the matching grade

In order to learn about each heuristic-attribute pair's contribution to the similarity grade, we repeat the comparison between the two executable files we used for examining the optimal weights on: version 0.3 and 0.7. When using the optimal weights we received that the max false-pos,flase-ng ratio is **0.023**. We repeated the test for each of the heuristic-attribute pairs, each time giving that pair zero for a weight. The results are shown below:

| Heuristic - Attribute | Min false-pos,false-neg ratios |
|---|---|
| ins_data_list | 0.084 |
| lib_calls_list | 0.031 |
| graph_rep | 0.028 |
| str_list | 0.026 |
| ins_type_list | 0.075 |
| imm_list | 0.088 |

Notice that the for all pairs, once removed, we get a min false-pos,flase-neg ratio higher than the one we received when we considered all the pairs.