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

By Yaron Kaner, Barak Sternberg and Zion Nahici

Reverse engineering (RE) is the process of discovering the technological principles of a device, object, or system through analysis of its structure, function and operation. Software RE involves taking a software program's machine code (the string of 0's and 1's that are sent to the logic processor) apart and analyzing its workings in detail to study how the program performs certain operations.

RE is a tedious process that takes a lot of time and dedication, as well as high familiarity with assembly language, compilers, and operating systems. During the past few years, many tools were made to make the reverse engineering process a faster one, and to lower the bar on the knowledge required to begin practicing it.

One of the problems that remain is identifying functions from the disassembled code according to their functionality - this still done manually today. Since code is often recycled, reused and rewritten, a lot of the RE effort is redundant. Many people have RE-ed the same pieces of code over the past years, without even realizing the potential of sharing their work with others. This project aims to solve this problem by allowing users to share their RE findings.

We've written a client-side IDA-Pro plug-in, and a complementary server. When installed, the plug-in offers two main functionalities: submitting the user's description and requesting public descriptions. Upon receiving a request, the server compares the user's function with the functions stored in its Database to find functions similar to the user's function.

After writing the plug-in we put a lot of work into making sure our product works – the functions the server decides are similar to the user's function are indeed similar.

Intro - Incentive

Reverse engineering (RE) is the process of discovering the technological principles of a device, object, or system through analysis of its structure, function and operation. **Software RE** involves taking a software program's machine code (the string of 0's and 1's that are sent to the logic processor) apart and analyzing its workings in detail to study how the program performs certain operations. This is done in order to improve the performance of a program, to fix a bug, to identify malicious content or to adapt a program written for use with one microprocessor for use with another.

A person practicing software RE may require several tools in order to disassemble a program. One tool is a hexadecimal dumper, which displays the binary code constituting a program in hexadecimal format (making it easier to read). Another common tool is the disassembler. A disassembler is a computer program that translates machine language into assembly language. Disassembly, the output of the disassembler, is often formatted for human-readability rather than suitability for input to an assembler.

The Interactive Disassembler, more commonly known as IDA, is a disassembler for computer software which generates assembly language source code from machine-executable code. It supports a variety of executable formats for different processors and operating systems. It also can be used as a debugger for certain executable file types. A decompiler plug-in for programs compiled with a C/C++ compiler is also available.

RE is a tedious process that takes a lot of time and dedication, as well as high familiarity with assembly language, compilers, and operating systems. During the past few years, many tools were made to make the reverse engineering process a faster one, and to lower the bar on the knowledge required to begin practicing it.

One of the problems that remain is identifying functions from the disassembled code according to their functionality - this still done manually today. Since code is often recycled, reused and rewritten, a lot of the RE effort is redundant. Many people have RE-ed the same pieces of code over the past years, without even realizing the potential of sharing their work with others. <u>This project aims to solve this problem by allowing users to share their RE findings.</u>

About the Plug-in

Our solution consists of two main parts:

1. A client-side Python plug-in for IDA.
2. A complementary Django server (hosting a database).

The two combined allow sharing of findings between those who practice RE.

When installed, the plug-in offers two main functionalities:

1. **Submitting** the user's description (function's name and comments).
2. **Requesting** public descriptions (submitted by other reverse engineers).

The client was written in Python and uses IDAPython, "an IDA Pro plugin that integrates the Python programming language, allowing scripts to run in IDA Pro".

The server is uses the Django Web framework and is also written in Python.

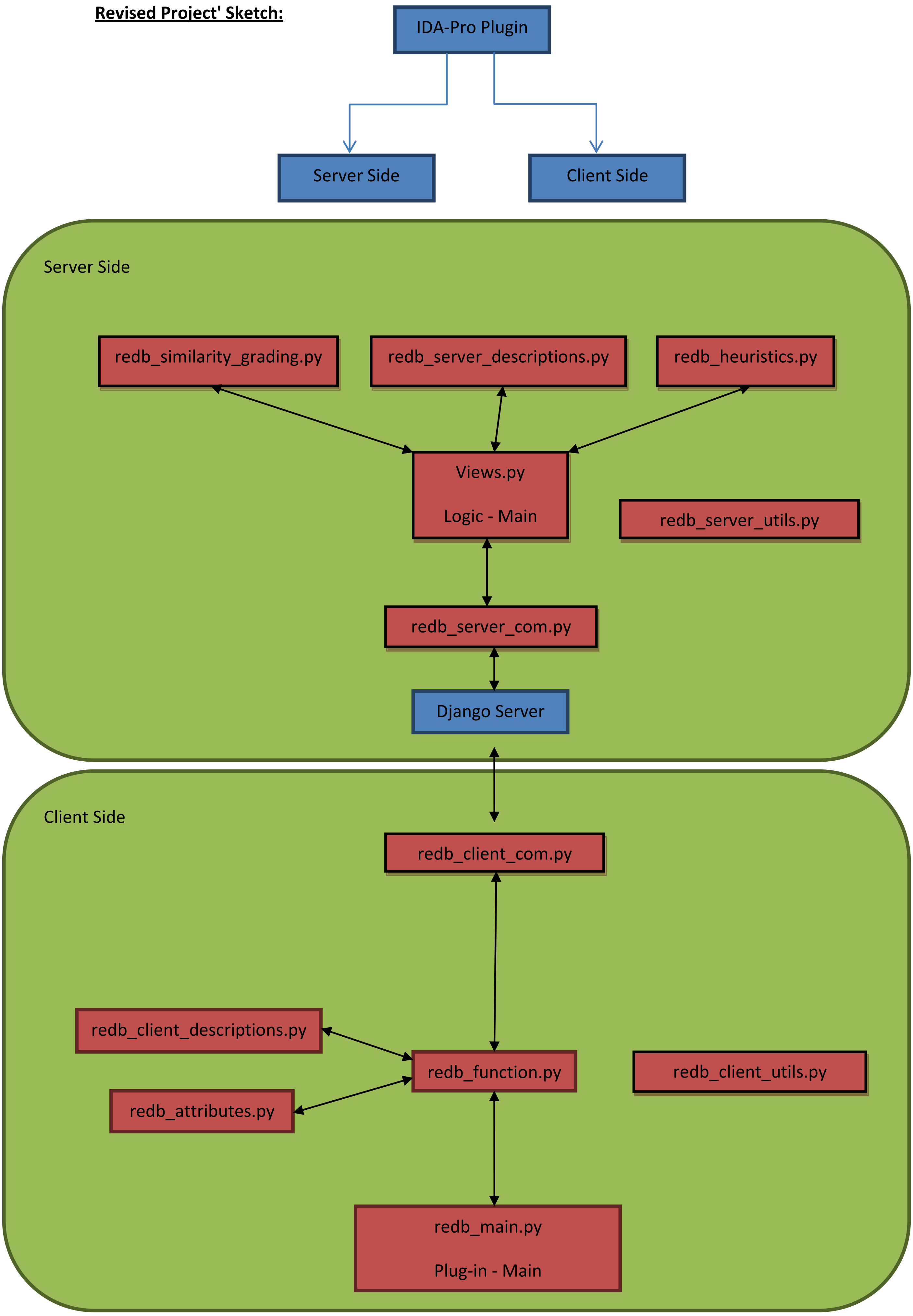
A difficulty we tried to tackle is that the same source code will compile differently into byte code, under different circumstances. The resulting compiled byte code depends on many factors, such as the compiler and its many configurations and the context in which the source code is found. The smallest change in the source code or in the compiler and its configurations will result in a different byte code, and therefore a completely different checksum. Therefore, a simple checksum of a function's byte code will not suffice for our needs.

Before executing both the Submit and Request functionalities, the plug-in will collect several **attributes** which characterize the function the user is working on. When submitting, these attributes are sent alongside the user's description, and when requesting descriptions, these attributes are sent to the server so it can look for similar functions.

Upon receiving a request the server compares the user's function with the functions stored in its Database. For each function in the DB it calculates a Matching Grade. This grade is simply a weighted average of grades, given to each pair of attribute instances (one attribute extracted from the user's function and one from the function taken from the DB). Descriptions for functions with the highest similarity grade will be returned to the user.

A pair of attribute instances is compared using what have defined in this project's context as **heuristics**.

Revised Project' Sketch:



Projects' Sketch Explanation:

Client side

- **redb_main.py** – the main module – maintains the plugin, handles user requests.
- **redb_function.py** – contains the REDBFunction class, which represents function, and contains all attributes, and descriptions.
- **redb_attributes.py** – contains the FunctionAttributes class which deals with attributes extraction for a specific function.
- **redb_client_descriptions.py** – contains the descriptions classes (Suggested and Public). A SuggestedDescription is a description which was received from the server. The server found it fitting for a specific function. This description is converted to a LocalDescription, and then displayed to the user upon request.
- **redb_client_com.py** – Methods and utilities required for communicating with the server.
- **redb_client_utils.py** – Utilities for all the other modules.

Server side

- Views - This module contains the server's Request, Submit and Compare handlers.
- redb_similarity_grading.py – Tools for Similarity Grading.
- redb_heuristics.py – Heuristics for comparing attribute instances.
- redb_server_descriptions.py – A description which was received from the server.
- redb_server_com.py - Methods and utilities required for communicating with the client.
- redb_server_utils.py - Utilities for all the other modules. Most are utility methods for the graph heuristics.

Heuristics and Attributes

Lists are compared using the List Similarity heuristic, **dictionaries** using the Dictionary Similarity heuristic, **graph representations** using the Graph Similarity heuristic, **integers** using the Integer Equality heuristic and **strings** using the String Equality heuristic.

Heuristics (in redb_heuristics.py)

1. **List Similarity** – Given two lists of objects, gives a grade to the two lists' similarity. Uses Python's SequenceMatcher.

2. **Dictionary Similarity** – Given two dictionaries of (object, number of occurrences) pairs, gives a grade to the two dictionaries' similarity. Uses to following algorithm:

```
1. Given two dictionaries A,B, such that:
    1. $A=\{(a_1,x_1),(a_2,x_2),\dots,(a_n,x_n)\}$
    2. $B=\{(b_1,y_1),(b_2,y_2),\dots,(b_k,y_k)\}$
    3. Where $a_1,\dots,a_n$ and $b_1,\dots,b_k$ are keys and $x_1,\dots,x_n$ and $y_1,\dots,y_k$ are values.
2. Define:
    1. $m:=|Union[\{a_1,\dots,a_n\},\{b_1,\dots,b_k\}]|$
    2. $\{c_1,\dots,c_m\}:= Union[\{a_1,\dots,a_n\},\{b_1,\dots,b_k\}]$
3. Compute:
    1. $d_i:=min[A[c_i],B[c_i]] / max[A[c_i],B[c_i]]$
    2. $f_i=min[A[c_i],B[c_i]] + max[A[c_i],B[c_i]]$
    3. $D:= d_1 f_1+ d_2 f_2+\dots+d_m f_m$
    4. $F:=f_1+f_2+\dots+f_m$
4. Return D/F
```

3. **Graph Similarity** – Given two graphs, gives a grade to the two graphs' similarity. See the Graph Similarity explanation below.

4. **Integer Equality** – Given two integers, simply checks if they are equal.

5. **String Equality** – Given two strings, simply checks if they are equal.

Attributes (in redb_attributes.py)

1. General Attributes
    1. Executable's Name - the executable's name.
    2. Executable's MD5 - the executable's MD5 signature.
    3. First Address - the function's first instruction's address in the executable.
    4. Number of Instructions - the number of assembly instructions in the function.
    5. Function's MD5 - the function's MD5 signature.
2. Instruction-related Attributes
    1. Instruction Data List – an ordered list of integers representing the instructions themselves.
    2. Instruction Type List – an ordered list of instruction types.
    3. Instruction Type Dictionary - a dictionary of (instruction type, count) pairs.

3. String-related Attributes
    1. String List – an ordered list of the strings which appear in the function.
    2. String Dictionary - a dictionary of (string, count) pairs.
4. Library Calls-related Attributes
    1. Library Calls List – an ordered list containing the library function names for library calls which occur in a function.
    2. Library Calls Dictionary - a dictionary of (library function name, count) pairs.
5. Immediates-related Attributes
    1. Immediates List – an ordered list of immediate values.
    2. Immediates Dictionary - a dictionary of (immediate value, count) pairs.
6. Control-Flow-related Attributes
    1. Graph Representation - a representation of the function's control-flow.

Matching Grade (redb_similarity_grading.py)

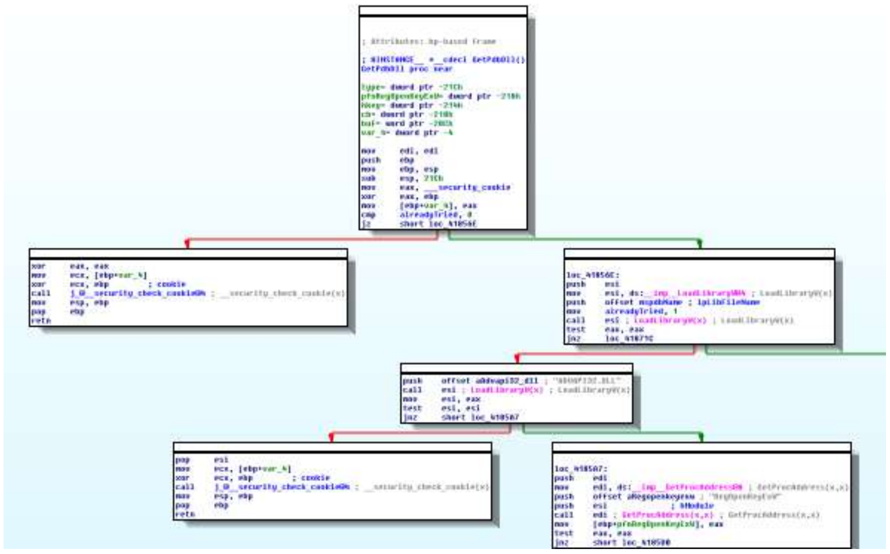
The server calculates the similarity grade for a pair of functions using several attributes. For each attribute, two of its instances are given as input to a heuristic, which in return outputs a grade. The final similarity grade is a weighted-average of the grades returned by the heuristics.

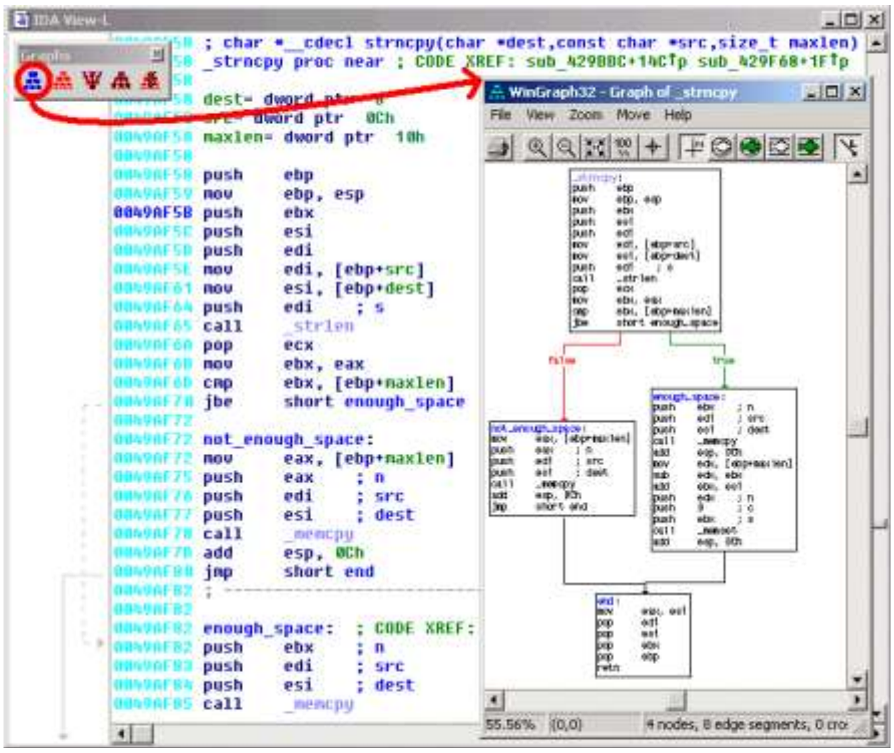
One of the questions that arose during the development is: what is the importance of each attribute when comparing functions – what weight should be given to each attribute. For further information see the **Finding the Optimal Weights** section in **Results**.

Graph Similarity

1. Introduction

Functions can be viewed as control-flow-graphs[1] using IDA. These directed graphs contain nodes, and edges between them, symbolizing various types of "jump" commands. IDA auto-generates this graph. Later on, functions are compared in the server based on their control-flow graphs, in a way which will be defined in this paper.





```
0049AF58 ; char *__cdecl strncpy(char *dest,const char *src,size_t maxlen)
0049AF58 _strncpy proc near ; CODE XREF: sub_4298BC+14C↑p sub_429F68+1F↑p
0049AF58
0049AF58 dest= dword ptr  8
0049AF58 src= dword ptr  0Ch
0049AF58 maxlen= dword ptr  10h
0049AF58
0049AF58 push    ebp
0049AF59 mov     ebp, esp
0049AF5B push    ebx
0049AF5C push    esi
0049AF5D push    edi
0049AF5E mov     edi, [ebp+src]
0049AF61 mov     esi, [ebp+dest]
0049AF64 push    edi     ; s
0049AF65 call    _strlen
0049AF6A pop     ecx
0049AF6B mov     ebx, eax
0049AF6D cmp     ebx, [ebp+maxlen]
0049AF70 jbe     short enough_space
0049AF72
0049AF72 not_enough_space:
0049AF72 mov     eax, [ebp+maxlen]
0049AF75 push    eax     ; n
0049AF76 push    edi     ; src
0049AF77 push    esi     ; dest
0049AF78 call    _memcpy
0049AF7D add     esp, 0Ch
0049AF80 jmp     short end
0049AF82 ; ---------------------------------------
0049AF82
0049AF82 enough_space:  ; CODE XREF:
0049AF82 push    ebx     ; n
0049AF83 push    edi     ; src
0049AF84 push    esi     ; dest
0049AF85 call    _memcpy
```

IDA View-L
Graphs
WinGraph32 - Graph of _strncpy
File View Zoom Move Help
false
true
55.56%
(0,0)
4 nodes, 8 edge segments, 0 cro

Given two graphs, we wish to quantify their similarity. Hence we define:

[1] http://en.wikipedia.org/wiki/Control_flow_graph

a. G1 and G2 are graphs. Let G3 be the largest graph (with regard to the number of nodes) which is a sub-graph of both G1 and G2.
b. Their similarity grade is given by defining their "Jaccard index"[2]:

$$\frac{|G1 \cap G2|}{|G1 \cup G2|} := \frac{|G3|}{|G1|+|G2|-|G3|}$$

Therefore, we need to find G3(only its size), a maximal sub-graph of both G1 and G2.

We accomplish this using "**Heuristics for Chemical Compound Matching"**[3]. We made slight modifications to it so it would fit our needs.

In order to maintain graphs in Python, we use an already-implemented package, NetworkX[4], for Python 2.7. It is used in the server side only, so the user won't have to install it.

2. Heuristics

2.1 As-Is Heuristic

(Implemented in redb_heuristics.py in the EqualityGraphComp class)

IDA creates the same graph (with the same node numbers and the same edges) for functions which are exactly the same, or differ in aspects not relevant to the control flow. This happens, for example, when the same function is used in two consecutive versions of the same software. The function's location in the binary, or its strings and immediate values may have changed, but its control flow remains the same.

In this heuristic, we will compare a pair of graphs.

Given two graph representations (list of edges) we check if their representation is exactly the same, i.e. check the lists are equal in the "==" sense. If so we return 1, otherwise we return 0.

This heuristic is inexpensive when compared with others, but when two graphs differ even slightly, it will return 0. So what we really check here is if the two graphs are equal, and not how similar they are.

[2] http://en.wikipedia.org/wiki/Jaccard_index
[3] jsbi.org/pdfs/journal1/GIW03/GIW03F015.pdf
[4] http://networkx.lanl.gov/

2.2. Chemistry Heuristic

(Implemented in redb_heuristics.py as NormalComp class)

Implements the "Jaccard" index for the two graphs compared.

In this heuristic, we will compare a pair of graphs.

The 'Heuristics for Chemical Compound Matching" article suggests a reduction to max-clique problem. First, given $G_1= (V_1, E_1)$ and $G_2= (V_2, E_2)$, we generate a new graph, called the "Association-Graph" (AG):

1. $V_{AG}=V_1XV_2$

2. Its group of edges is defined in the following way:

> (t_1, j_1) and (t_2, j_2) are adjacent in AG iff ((t_1 and t_2 are adjacent in G_1 and j_1 and j_2 are adjacent in G_2) or (t_1 and t_2 are not adjacent in G_1 and j_1 and j_2 are not adjacent in G_2), with the exception that when $t_1 == j_1$ or $t_2 == j_2$ then edge between them is not possible (we wish the pairings to be injective).

We want nodes in the AG to symbolize pairing of two nodes in the original graphs (one from each of the two graphs). The connection to the sub-graph lies in the demands on the edges: cliques in the AG correspond to common sub-graphs of G1 and G2. Moreover, it can be proven that there exists a bijection between the cliques of the AG and the common sub-graphs of G1 and G2.

Hence, we only need to find the maximal-clique size in the AG, which equals the maximal size of a common sub-graph of G1 and G2. This is done using the already implemented algorithm for finding max-clique-size in NetworkX.

This heuristic is very expansive and inefficient, especially when the graphs are large. For further information, refer to the article.

2.3. ListGraph Heuristic

(Implemented in heuristics.py as ListGraphComp class)

The Chemistry Heuristic is not fast enough for practical uses (NP-complete problem) when it comes to dealing with larger graphs. Most functions contain "choke points", i.e. nodes which every path from the first instruction of the function to one of its ends must pass them. They split the function into smaller consecutive blocks. The execution of such functions is equivalent to the serial execution of these blocks. Therefore we can treat the graph as a list of smaller graphs.

In this heuristic, we will compare a pair of ListGraph objects. ListGraph is defined below and saved in the server's DB during submission.

A "thin edge" is defined to be an edge that each path from the root of the graph (the first instruction of the function) to the target (one of the function's return instructions) passes through it. **A "thin node"** is defined to be a node that each path as described above passes through it.

First, the method will create a new list of graphs from a given graph. It goes through the following steps:

1. **Collapsing:** "collapses" the graph- i.e. deletes the spare edges, the edges that their deletion won't affect the control flow graph. Deletes the edge *(a, b)* between nodes *a* and *b*, if the edge is the only edge going out of *a*, and the only going into *b*.
2. **Condensing:** after the deletion of the edges, it condenses the graph (condensation to "strictly connected components graph"). This is done to improve graphs' structure, splitting it to "loop-blocks" as well as "basic-blocks".
3. **Expanding:** expands the graph so that for each node neither the in-degree nor the out-degree is equal to 1. It does so by expanding the node into two new nodes – their predecessors and successors are those of the original node. This is done to ensure that if a node is one of the "thin-nodes", one of his edges will be a "thin edge".

Creating the GraphList:

We start by finding the graph's "thin-edges", and then splits the graph by these edges. Given a list of bridges, the splitting of the graph done by:

a. Finding topological sorting of the graph (the graph is condensed, and therefore a DAG).

b. The list of "thin-edges" is sorted according to a topological sorting of the graph (there is only one way to sort the list of the "thin-edges", since they are "thin-edges", and there is only one order they can passed through). Given two adjacent "thin-edges" in the list, takes the graph that between the tail of the first "thin edge" and the head of the second "thin edge".

The list of the graphs collected is the list-graph we will compare.

2.3.1. Comparing a pair of graph-lists of the same size:

When the graph-lists are of the same size, the similarity grade between them, using the "Chemistry Heuristic", can be defines so:

1. Suppose, we compare two list graphs: $l_1 = [a_1,a_2,\dots,a_n]$, $l_2 = [b_1,b_2,\dots b_n]$, where a_i and b_i are graphs (for i in [1,…,n]).
2. Denote the similarity grade output between two graphs, a_i and b_i, using the "Chemistry Heuristic" as $CH(a_i,b_i)$.
3. Then, total-similarity grade between them will be:

$$\frac{\sum_{i=1}^{n} CH(a_i,b_i)*(|a_i|+|b_i|)}{\sum_{i=1}^{n}(|a_i|+|b_i|)}$$

   What it does is to give weights (the graphs' total size) to grades of similarity between graph elements in the graph-lists.

2.3.2 Comparing a pair of graph-lists of different sizes:

Given two lists of graphs, it tries to embed the smaller list in the large one. It runs on all the possible ways of embedding, and chooses the one that the method above gives the optimal grade for (using comparison of graph-lists with the same-size).

3. Graph representation attribute:

On the client-side, only the regular representation of the control-flow-graph is computed. After being submitted to the server, the graph attribute is saved as three seperate graphs: 'normal_graph', 'compressed_graph' and 'list_graph'.

1. ***'normal_graph'*** - the original representation of the graph as it was sent to the server.
2. ***'compressed_graph'***: the compressed representation of the graph (after: 1. collapsing nodes, 2. removing unreachable nodes, 3. and conversion to strictly connected graph).
3. ***'list_graph'***: the list graph.

4. Final grade of Graph Heuristics

These constants are defined on server side:

- Const1 :=MAX_GRAPH_COMP_SIZE
- Const2 := MIN_MG_GRAPH

How the final grade is given for two graphs, G1 and G2:

1. **Checking for minimal number of nodes**: if one of the graphs (in their 'compressed_graph' representation) does not have enough nodes (Const2) – graphs won't be compared and this grade won't be used in the total matching grade.
2. If **G1 and G2 are identical using "As-Is Heuristic"**, then return **1.0**.
3. If **$|G_1|*|G_2|$ < Const1**, then return **"Chemistry Heuristic" between G1 and G2**.
4. If **$|G_{1,compressed}|*|G_{2,compressed}|$<Const1**, then return **"Chemistry Heuristic" between $G_{1,compressed}$ and $G_{2,compressed}$**.
5. Using their 'list_graph' representations, **compare G1 and G2 using "ListGraph Heuristic"**.

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.

The optimal weights

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.

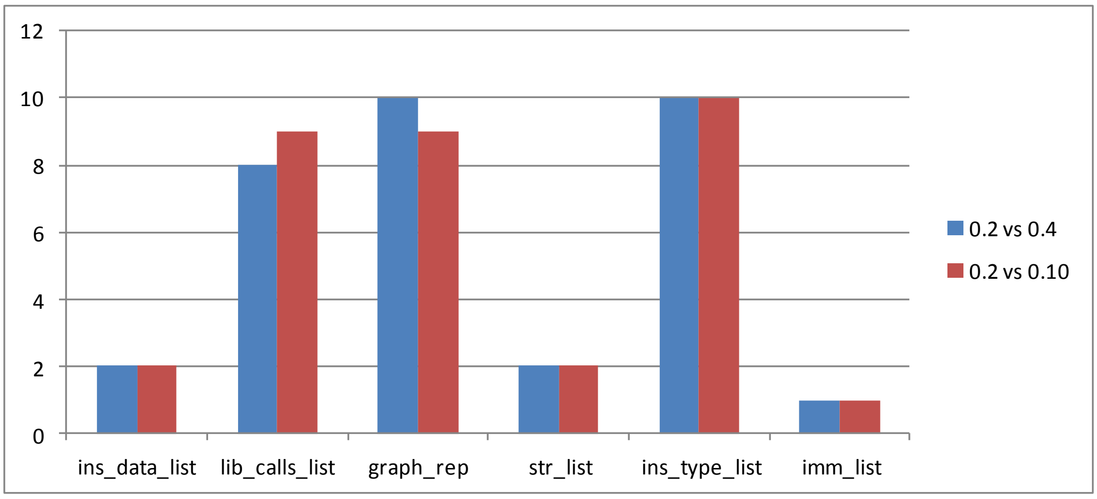
The heuristic-attribute pairs, in the order in which their grades appear in the table:

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:



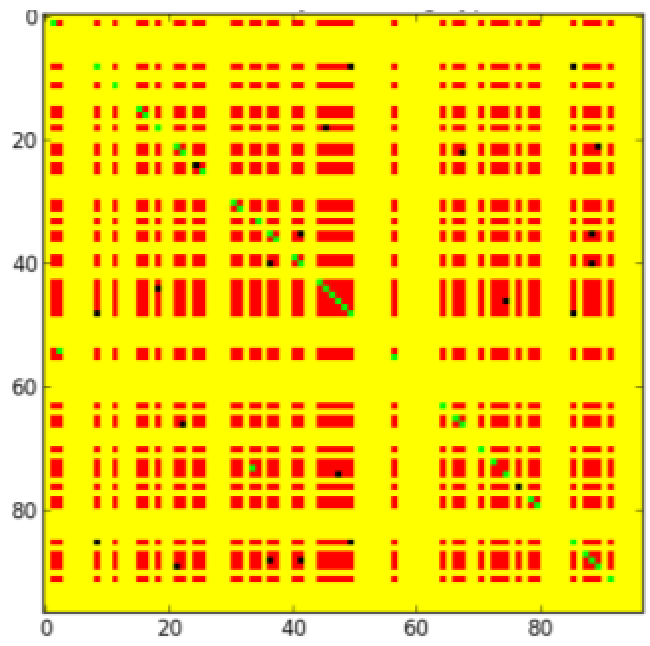
- Averaging the results, the following weights are received:
    - ins_type_list – 10
    - graph_rep – 9
    - lib_calls_list – 9
    - ins_data – 2
    - string_list – 2
    - imm_list – 1
- When we compared executable files from other applications, some pairs received different weights. For example, 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:

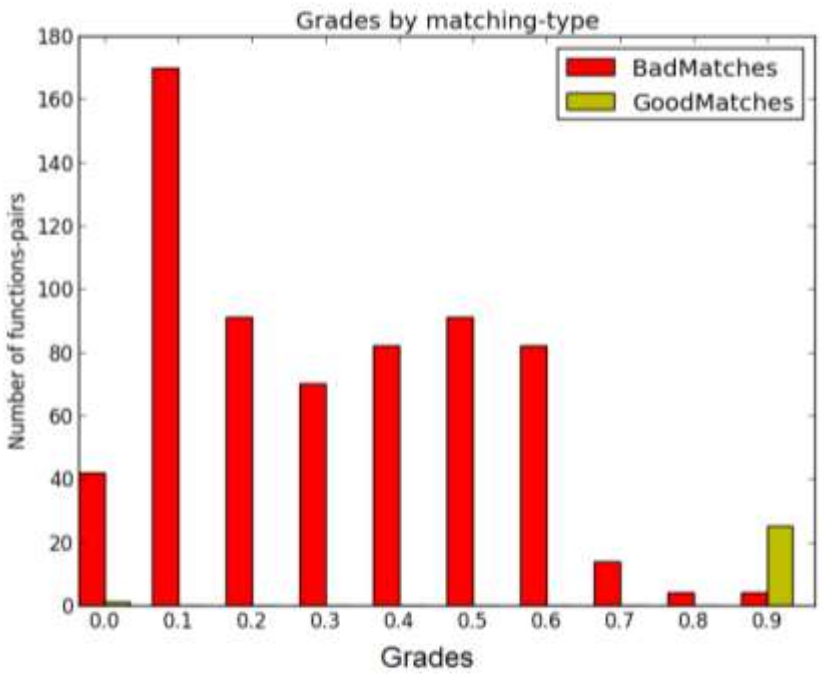
1. Success/Failure Graph – closely related to the two dimensional array of similarity grades discussed before. The pivot is located at the top left corner. Both axes represent functions from an executable file, each a different file. A function has a place on one axis if a function of the same name exists 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.



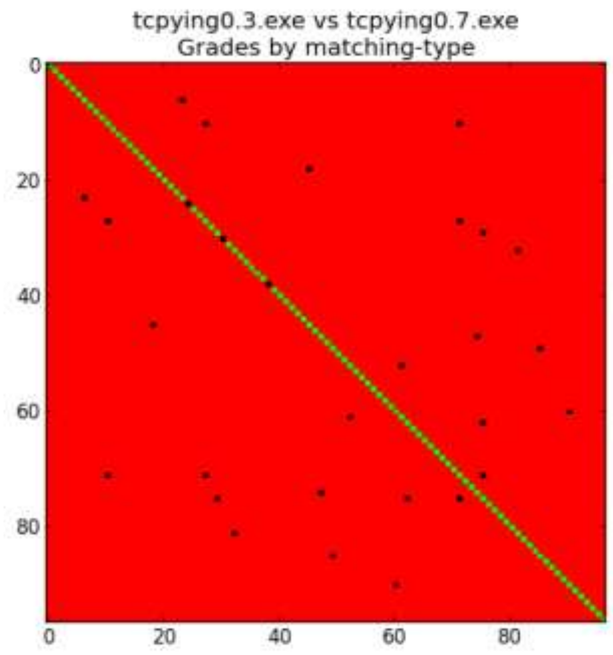
2. Grade Distribution – what grades were computed for a set of tests? Good matches refer to comparisons in which the two functions share the same name. Bad matches

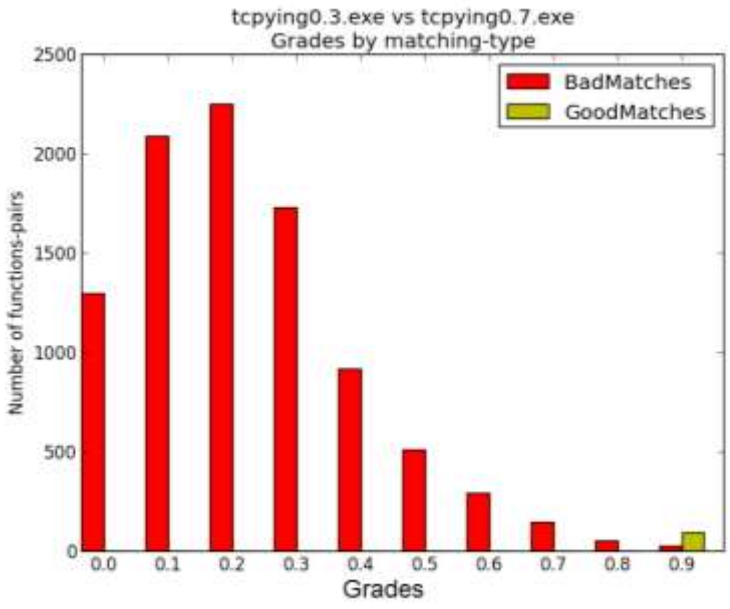
refer to comparisons in which the two functions have different names.



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

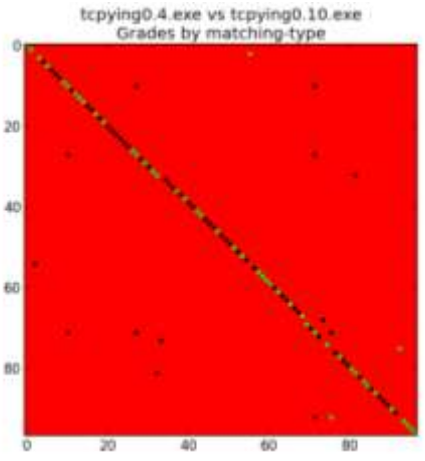
We wish to examine each heuristic-attribute pair by itself. We do this by repeating the tests we have done thus far for the matching grade, except give a weight to one of the pairs at each time (all the others get zeroes as weights).

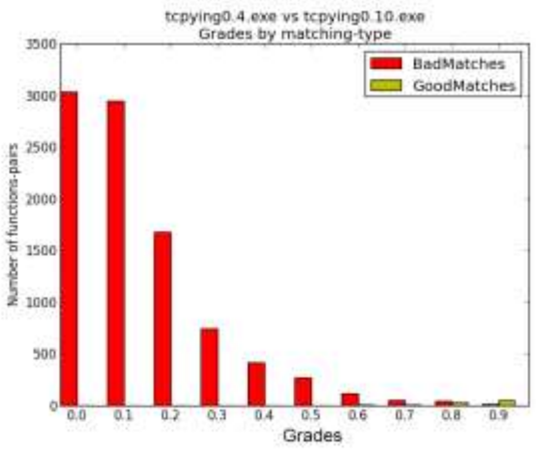
The results are shown below:

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

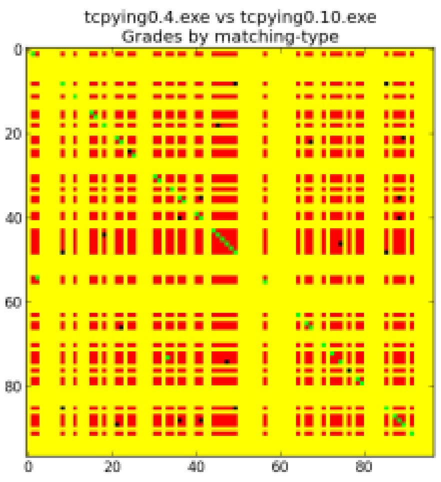
The matching graphs:

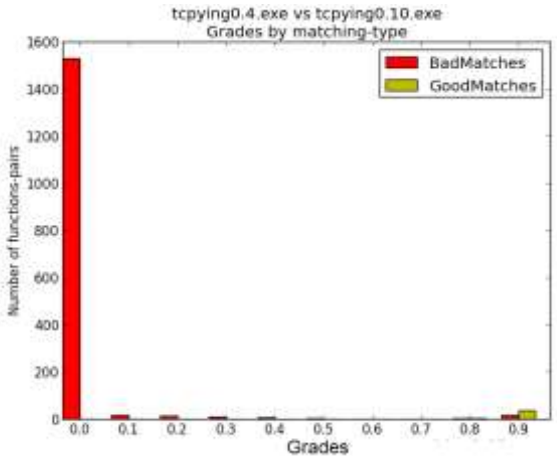
ins_data_list



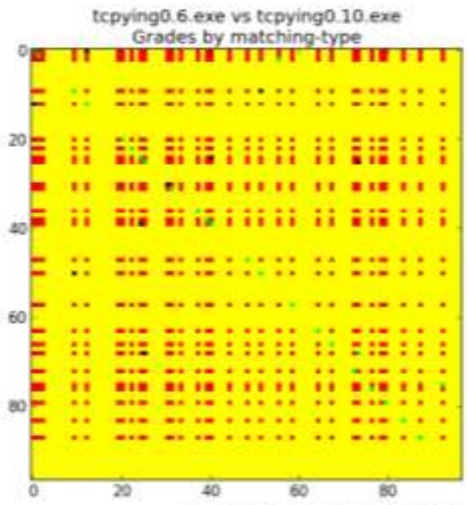


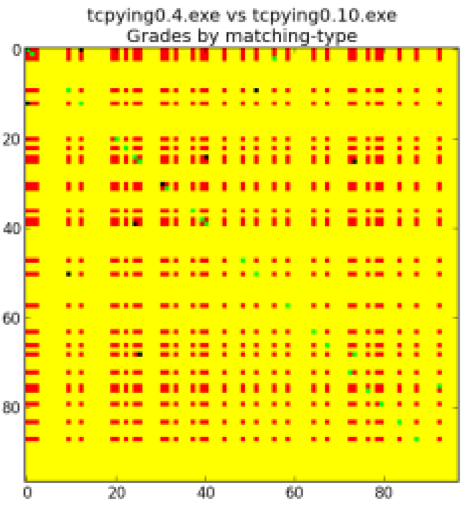
lib_calls_list

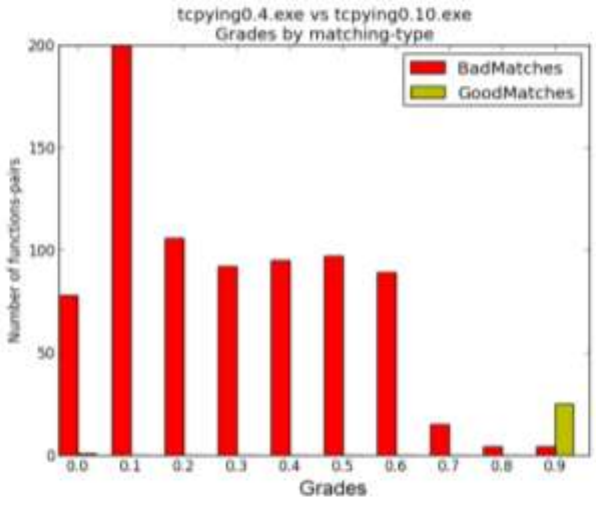




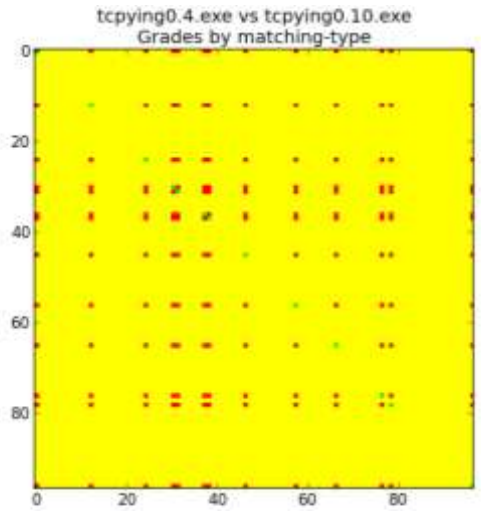
graph_rep

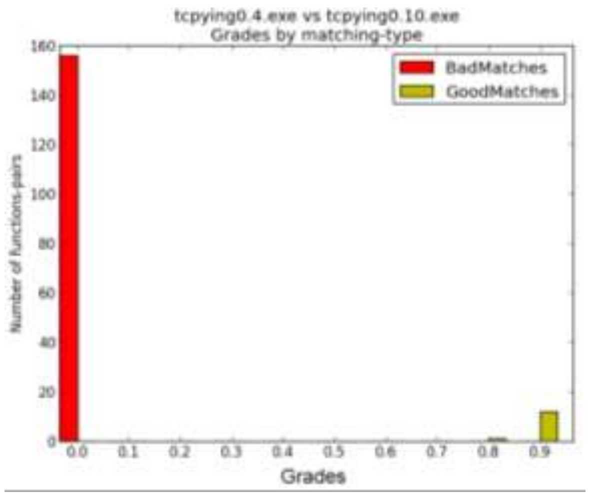






str_list

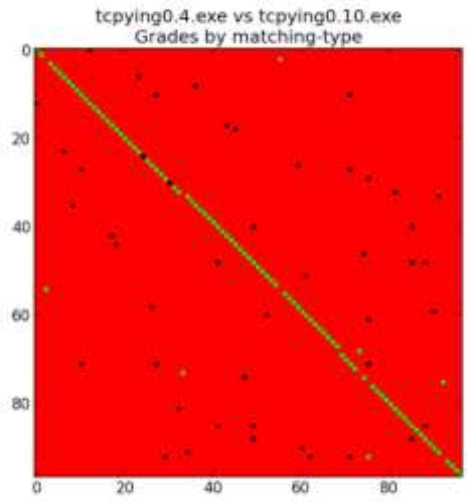


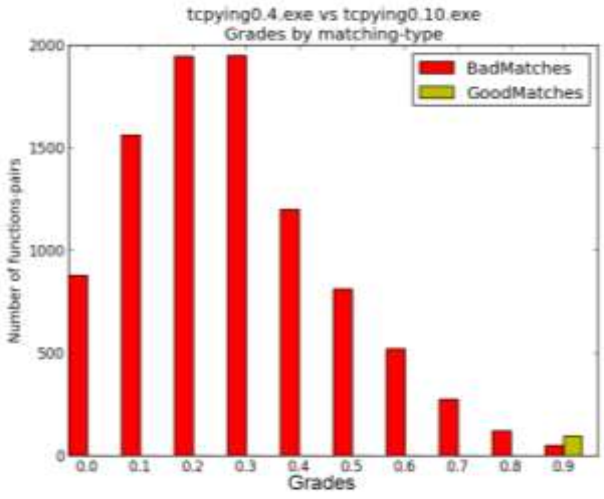


Note the almost-perfect separation between pairs of matching functions and pairs of non-matching functions. This is due to the fact that it is highly unlikely for two non-similar functions to have common strings.

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

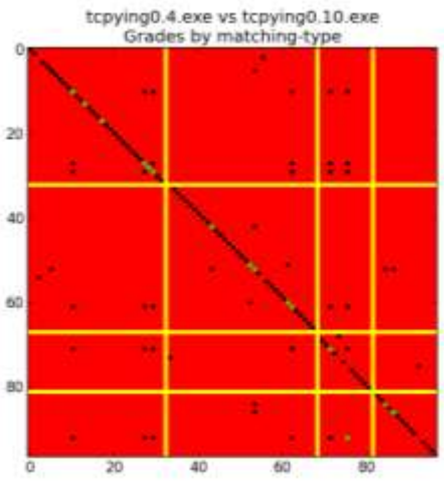
ins_type_list

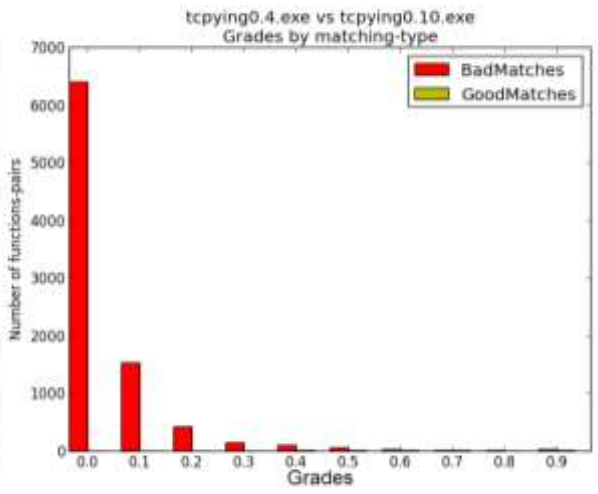




As can be expected, there are no yellow pixels, since every function contains instructions and thus can be compared by their types.

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

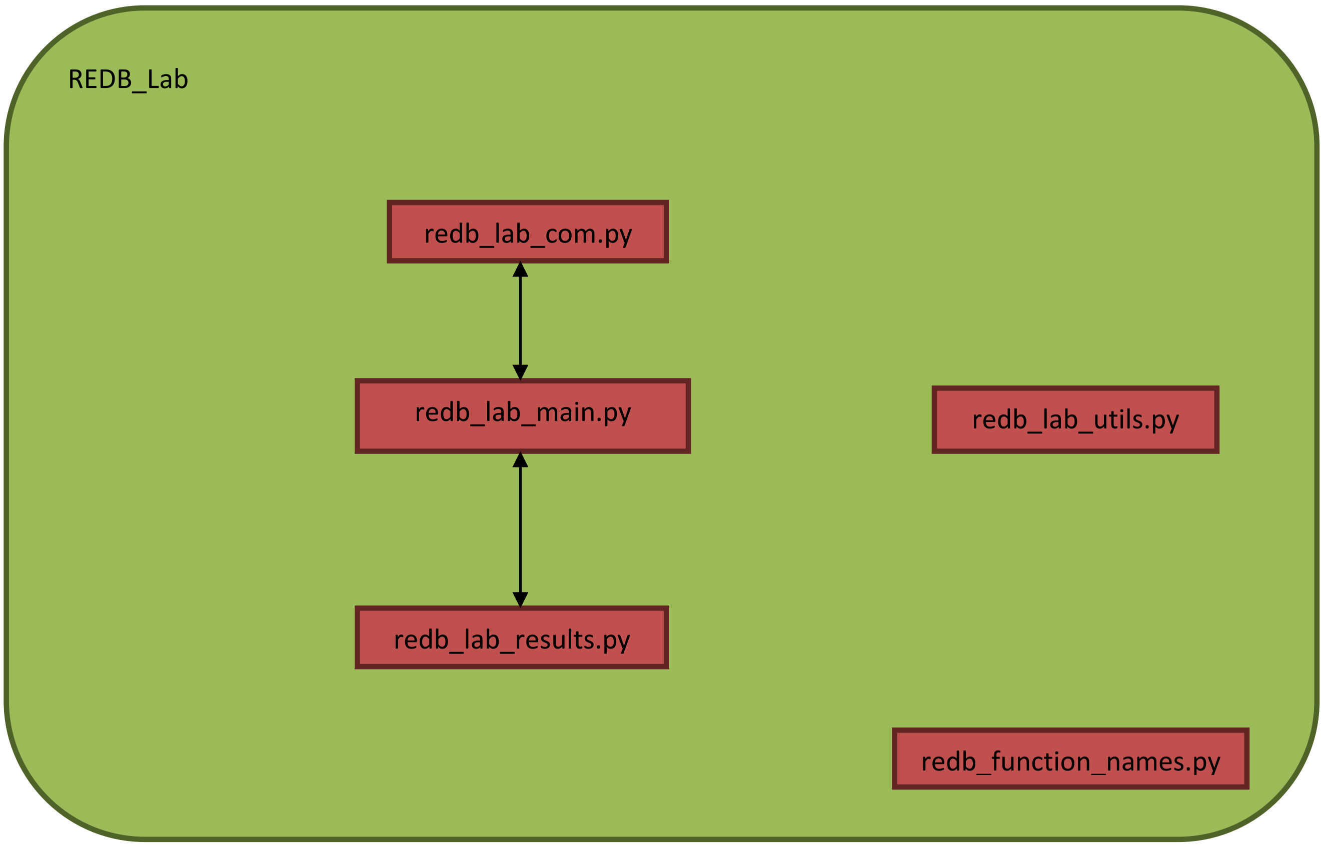
In order to learn about each heuristic-attribute pair's contribution to the similarity grade, we repeat the comparison between the two executable files we used for examining the optimal weights on: version 0.3 and 0.7. When using the optimal weights we received that the max false-pos,flase-ng ratio is **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.

REDB Lab:

REDB_Lab was used for testing our projects and generate all results.



redb_lab_main.py is responsible for communicating with the server, and running queries on it. There is a special URL you can use to query the server. The user can run queries after he had already uploaded functions to the server. Abilities:

1. Query the server with any possible comparison.
2. Compute custom matching grades.
3. Drop the results to the disc in several formats.
4. Compute measures to a set of comparison results (see more in the results section).

redb_lab_utils.py is used by RunQuery.py. Responsible for processing received data from the server. In addition, it calculates total matching grades, for some compound of heuristics grades, using specific weights or finding optimal weights (the ones which give maximal grades for the measurements we have defined).

redb_lab_results.py: responsible mostly for calculating measures for sets of comparison results.

redb_function_names: A script meant to be run within IDA, and used to extract function names from one executable or the union/intersection of a few executables functions names.

More information regarding this package is available in the results section.

Plans compared with results

Project Plan:

Compared to the project plan, we have followed it almost completely. We haven't implemented the ranking system which was as it was defined as a nice-to-have feature. We were advised not to waste time on this functionality, as there were features of greater importance.

As the project advanced and the image became clearer, we added more specific designing details.

We can now check the current status of goals we set ourselves in the milestones:

Milestone 1:

Mission: Adding basic properties to the data sent and received from client to server: repeatable and normal comments, date of upload, the name of the function.
Status: Done.
Description: all function' description object is extracted for each handled function. This object contains all the data mentioned in the mission.

Mission: Dropping the relativity factor (variables and addresses) (relativity attribute).
Status: Done
Description: we have added attribute which extracts all instruction-types for handled function. It is called "_ins_type_list", it doesn't collect parameters. And therefore, omits all extra-data.

Mission: Coping with the "multiple comments/descriptions per function" aspect: Both in the server and client side.
Status: Done.

Mission: Adding simple heuristics and attributes
Status: Done.
Description: all heuristics and attributes have been implemented.

Milestone 2:

Mission: Similarity with regard to order of appearance.
Status: Done.
Description: added attributes which are saved in an ordered list. The list is then compared using the "sequence-matcher" with the ListSimilarity Heuristic.

Mission: Expanding and fine tuning custom similarity.
Status: Done.
Description: tested many executables using the **REDB_Lab** package, thus finding optimal weights for the server's total matching grade.

Mission: Adding configurability: Run on demand / on executable load.
Status: Done.

Description: the plugin runs on each executable loaded using IDA.

Mission: Adding configurability: Query configurations.
Status: Done.
Description: added options for user, can now select how many descriptions will be returned by the server.

Mission: Adding configurability: User details.
Status: Not implemented.
Description: low-priority.

Mission: Adding support for different IDA versions.
Status: Partially.
Description: support was tested and worked only in IDA 6.3/6.1.

Mission: Function's automatic name-changing.
Status: Done.

Mission: Function graphs similarity ranking / heuristic.
Status: Done.
Description: Added new heuristics and attribute for graphs comparison.

Mission: Embedding descriptions within the code.
Status: Done.
Description: we have improved the embedding the descriptions within the code. All descriptions now have a property of "can_be_embedded", and if a description can't be embedded, a short version of it is displayed.

Mission:

a. "Request descriptions for all functions" option.
b. User chooses functions manually / Ran automatically on all available functions.

Status: Changed, Done.

Description:

a. After using the plug-in for a while, we have decided that submitting/requesting **all functions' data** is ineffective (**although we have implemented this functionality for our tests).**
b. Therefore, the user now has to ***handle*** functions before submitting\requesting them from the server.