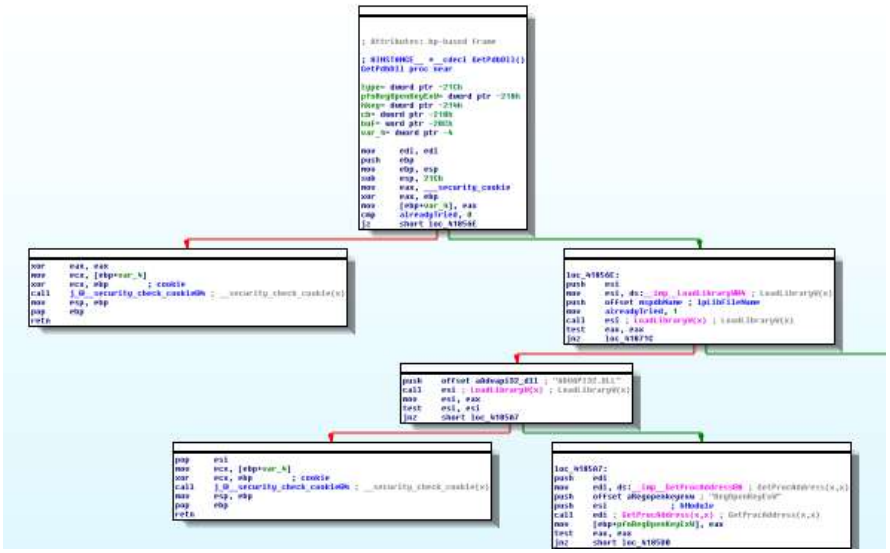
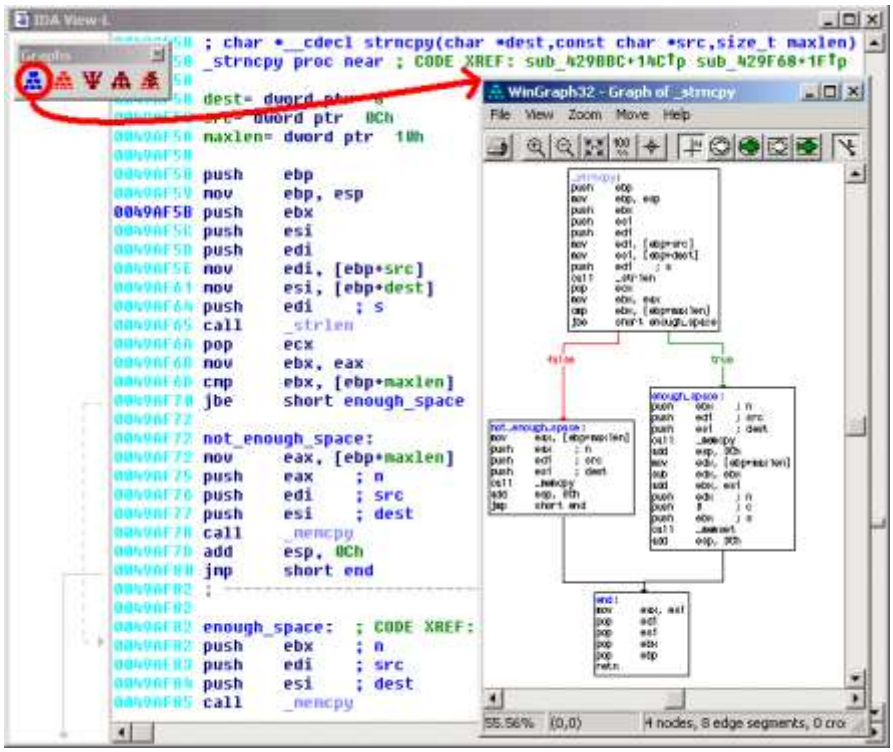
Reverse-Engineering Database – An IDA-Pro Plug-in

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.





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

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

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. $V_{AG} = V_1 X V_2$

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: $l_1 = [a_1,a_2,\ldots,a_n]$, $l_2 = [b_1,b_2,\ldots b_n]$, 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:

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

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"**.