Network-based library detection in compiled code for software reverse engineering

Jordan Rabet
Stanford University
jrabet@stanford.edu

I. Introduction

Software reverse engineering is the practice of taking apart compiled software in an effort to understand how it works. It has several applications, for example being an especially important aspect of vulnerability research, as it is often necessary to reverse engineer closed pieces of software to find vulnerabilities, or reverse engineer pieces of malware to understand which vulnerabilities they exploit. It is also a difficult and long process as the amount of code which needs to be analyzed is considerable, and there are typically no debug symbols available. One way of making this process easier is to automatically port findings from one code binary to another : for instance, two code binaries might use the same libraries, and so if we can automatically find a mapping between the library functions in one binary with the corresponding ones in the second one, we can reduce the amount of work necessary and streamline the process. In practice, this sort of situation can occur in a variety of situations : different applications running on a given embedded platform will likely share a statically-linked runtime library, and pieces of software which receive updates will obviously share large portions of code across across versions. In these cases, being able to port over findings from one binary to another with minimal effort is essential.

There are several issues which arise when approaching this task. Mostly, there is considerable variability which can be introduced by different compilers (or simply compiler options) in the compiled code, which typically makes naive binary data comparison ineffective. Our observation is that we can statically extract a function call-network from a compiled binary, that this network should be largely invariant to compilation variability, and that structure within that network can be used to identify parts of libraries. Using this information, we explore ways of efficiently matching functions from a query call-network with those in a target call-network, in the hopes of annotating as many as possible.

A. Previous work

Tools to achieve this task already exist. Most notably, Hex-Rays' Interactive Diassembler software includes "Fast Library Identification and Recognition Technology" (FLIRT for short), which works by generating "signatures" from the binary data of compiled functions, and then uses an efficient algorithm to both store and match them with newly detected functions in other binaries [4]. The approach is generally effective but is prone to failing if the target binary was generated by a different compiler, or simply using different compiler optimization options. Additionally, it does not take into account the relationship between functions which call one another.

Given our network-based approach, the problem we are trying to solve can essentially be seen as a specific form of the subgraph isomorphism problem : we have a collection of annotated functions from our source network, and we want to find their occurences in the target graph. In the general case, the subgraph isomorphism problem is a notoriously difficult one to solve efficiently, as it is NP-complete [1]. In our case, being able to find subgraph occurences efficiently is important as we will potentially have hundeds of functions to match in networks of thousands of nodes, scales which make factorial time algorithms impractical. One such factorial time subgraph isomorphism algorithm is VF2 [2], which essentially works by defining a state space of potential solutions and then traversing that state space as efficiently as possible to find all subgraphs matching a query function network in the target. The algorithm achieves a $O(N!N)$ worst-case time complexity, and performs best on modestly-sized networks.

There are many graph and subgraph isomorphism algorithms which were developed for special-case networks. For example, Aho, Hopcroft and Ullman proposed a linear-time graph isomorphism algorithm for trees [3] (referred to in the following as the AHU algorithm). While at first glance our problem is closer to the subgraph isomorphism problem than the graph isomorphism one, and while the network we consider is not necessarily a tree, it might be possible to make use of such results to develop an efficient algorithm specific to our problem.

II. Problem definition

A. Call-network

We define a program's call-network as being the network where each function is represented by a node, and each function call is represented by a directed edge, such that if a program has two functions A and B, where A calls B once, then the corresponding call-network will be contains two nodes A and B and a single edge $A \to B$. Due to the nature of the network, there can be multiple identical edges (if A

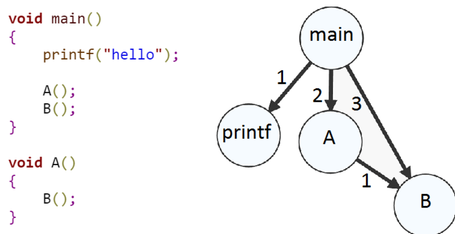

Fig. 1. Left : a sample toy C program. Right : the corresponding call-network. The edge labels indicate the call order. .

calls B twice, we would have two $A \rightarrow B$ edges), and edges can be labeled with the order in which calls happen in the caller function. A sample call graph on a toy C program can be found in Figure 1.

This definition of the call-network leads to a nummber of properties of all call-networks. Mostly, we can say that call-networks are similar to trees, though they do distinguish themselves in important ways :

- Like trees, all call-networks have a single root node : the program's entrypoint function.
- Like trees, we can visualize the nodes that a given node points to as its children.
- Unlike trees, call-networks may contain cycles, though they are relatively rare. For example, recursive functions will be associated to self-pointing edges.
- Unlike trees, call-network nodes can have multiple parents.

We define the height of any node in a call-network as the longest directed path from it to a leaf.

B. Library-matching

Our goal is to find an efficient algorithm which makes use of the available network structure within parts of the source network which correspond to library functions to find the corresponding functions in a target network. Formally, we can define the source call-network as S and the target call-network as T. S is annotated, which means that any number of nodes it contains is labeled which information we want to transfer to T. We define L a set of nodes within S which correspond to library functions which we want to identify in T. By joining the nodes in L into connected components, we can define a set of subgraphs $\bar{L}$ of S. Because of the fact that the nodes in L correspond to library functions, we know that they are "terminal" subgraphs, meaning that any leaf within a graph belonging to $\bar{L}$ is also a leaf in S. This property is due to us considering that library functions will only statically call functions from its own library or other libraries, but not user code. While there are cases where this is not true (for example, overridable library functions), they are few enough that we can ignore them for our purposes. Our goal is then to find an efficient algorithm for finding subgraphs of T which are isomorphic to graphs in $\bar{L}$, and which share the property of being terminal.

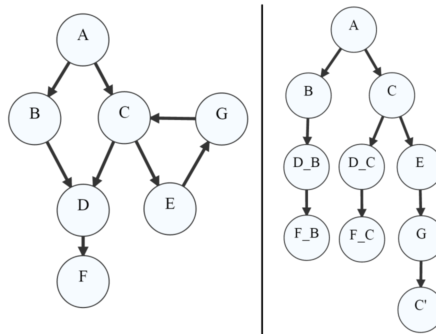

Fig. 2. Left : an example toy call-network. Right : the corresponding transformed call-network. The cycle list in this case would be $[(C, C')]$, while the multiple-parentage list would be $[(D, D_B, D_C)]$. .

III. Call structure matching algorithm

Making use of the properties we've just explained for both call-networks in general and library subgraphs, we can devise an efficient algorithm which reduces the problem to performing tree isomorphism checks on a limited number of candidates. Doing so takes a two-part algorithm : first, we need a way to reduce the isomorphism checks on library subgraphs to the tree-subgraph isomorphism case, and then we need to efficiently select candidates in the target network for each library subgraph in the source network.

A. Library subgraph tree transformation

We first define a transformation from a subgraph into a form where isomorphism can be established efficiently using a tree isomorphism algorithm. Given the tree-like structure of the graphs, this is a simple process : we just need to take care of non-tree properities. This is done by creating a new graph which starts out identical to the input, but is then modified in two ways :

- For each cycle in the graph, the last edge $B \rightarrow A$ in the cycle is removed and replaced by a new edge $B \rightarrow A'$ to a new node A'. The couple (A, A') is then added to a list so that the cycle's existence can be kept track of (this will be necessary for establishing isomorphisms).
- For each node A in the graph such that there is more than one node B for which there exists an edge $B \rightarrow A$, the subgraph rooted in A is duplicated into a subgraph rooted in a new node A_B. The original edge $B \rightarrow A_B$ is then replaced with a new edge $B \rightarrow A_B$, and the tuple $(A, A_B, ...)$ is added to a list to keep track of the original structure (this will be necessary for establishing isomorphisms).

An example of this transformation can be seen in Figure 2. Given this, checking for isomorphisms between two subgraphs is simple : first apply the transformation to both, then check if the resulting trees are isomorphic using an efficient algorithm,

and if they are then compare the lists of cycles and of multiple-parentage to make sure that the original graphs are indeed isomorphic. The transformation itself can be done in linear time over the size of the transformed tree, as can the isomorphism check, and the cycle and multiple-parentage checks. The size of the transformed tree depends on how many cycles and multiple-parentage nodes the source graph contains, but on average we can say that this number should be small compared to the number of nodes in the graph. Therefore, we should be able to check for isomorphism between library function graphs in almost linear time on average.

B. Target subgraph candidate selection

Given the ability to check isomorphism on given pairs of call-network subgraphs, the task remains of trying to find as few matching candidates in the target graph as possible, and efficiently. To do so, we take advantage of two structural invariants :

- We know that library subgraphs will be rooted in a single node, and that they are terminal, as explained previously. Given this, we know that the height (that is, as defined previously, the length of the longest path to a leaf) of the roots of two library subgraphs must be equal for them to have any chance of being isomorphic.
- For any two subgraphs to be isomorphic, they must have the same number of nodes. We can therefore avoid considering subgraphs which do not meet this criteria.

Given a node, we can easily consider the subgraph which will have it as its root. Therefore, we can also easily consider the number of nodes of that subgraph, as well as its height. In fact, we can compute both of those quantities in linear time over the number of nodes in each graph, using a simple depth-first traversal algorithm. Given this, we can store sets of nodes in two tables : one whose index is the number of nodes of its subgraph, and one whose index is the height of its subgraph. Given these, as well as a query library subgraph, we can find all candidate subgraphs simply by taking the interesection of the corresponding sets in both of those tables. Indexing is done in linear time, while intersection can be done in linear time over the sizes of the sets, given a proper data structure.

C. Full algorithm

Given these pieces, putting the algorithm together is fairly simple : we maintain a list of query library subgraphs sorted by height, and as long as that list is non-empty, we take the subgraph with the largest height value, try to match it to a subgraph in the target network, and if we fail we remove its root and add all of its child subgraphs to the list. Doing it in order of descending subgraph heights is meant to ensure that bigger heights have priority, to avoid subgraphs with smaller heights (and therefore most likely less exploitable structure) from being detected as part of what should actually be matched to a taller subgraph. The list of query library subgraphs can be initialized in one of two ways :

- It can be initialized to $[0]$, meaning we attempt to match the entire source call-network as a library subgraph. Of course this will fail, and the graph will be automatically subdivided into ever smaller subgraphs, which will eventually all be matched and cover all libraries.
- It can be initialized to a manually selected list of library functions.

A pseudocode version of the algorithm can be found in Algorithm 1.

Algorithm 1 Algorithm for matching library functions in call-networks

Input: S, T, and $\bar{L}$
Output: matches

```
compute_height_subgraph_size_tables(S)
compute_height_subgraph_size_tables(T)
Q ← L̄                         ▷ Q is a priority queue over height
matches ← ∅
while |Q| > 0 do
    g ← pop(Q)
    t_g ← tree_transform(c)
    h ← height(g)
    s ← subgraph_size(g)
    C ← subgraphs_height(T, h) ∩ subgraphs_size(T, s)
    i ← false
    for c ∈ C do
        t_c ← tree_transform(c)
        if is_isomorphic(t_g, t_c) then
            matches ← matches ∪{(g, c)}
            i ← true
        end if
    end for
    if not(i) then
        for k ∈ children(c.root) do
            Q ← Q ∪ k
        end for
    end if
end while
```

IV. Ambiguity resolution

The algorithm we just proposed will find matching call structures across two programs, which is likely to result in ambiguities, ie finding multiple functions in S which match a single function in T's call structure. This is problematic as we currently have no way to choose which match is most likely. There are many different situations in which this can arise; we list a few typical examples using standard functions :

- `sprintf` can be matched with both texttsprintf and texttsnprintf. Clearly, this would be due to the fact that texttsnprintf shares the exact same call-structure as texttsprintf. Given the nature of the difference between texttsprintf and texttsnprintf, it seems unreasonable to expect a purely network-based system to be able to distinguish them, unless we make assumptions about

S and T. (for example, if we assume that T is an updated version of S's software, then we might be able to disambiguate them)

- fprintf can be matched with both printf and fprintf. While this is similar to the previous case (they share the exact same internal structure), we might be able to disambiguate it through context : for example, it is likely that the parent nodes for fprintf will make calls to fwrite or fopen, while the parent nodes to printf have little reason to make similar calls. Due to this, by looking at sibbling nodes across S and T, we might be able to disambiguate these matches.

- in preliminary tests, we've seen ctrulib [9] (a platform-specific library) functions gfxSwapBuffers matched with both srvInit and gfxSwapBuffers. This is a case where the source function has very little structure to work with, and where by chance another function shares it, even though they are completely unrelated and do not share actual functions. While it is difficult to make any guarantees in this case due to lack of information, it might be possible to make use of extra information from the rest of the network in the same way as proposed in the previous case.

We believe that we can deal with this issue by scoring potential matches according to features, and using those to compute a likelihood score for each match.

A. Feature framework

We first define what a feature is in this context and how it will be used. We define a feature F as being a function which takes a couple (A, B) of nodes in S and T respectively as input and outputs a score in $\mathbb{R}^+$. We define a set of features $(F_i)_i$ which we will apply to our matches. Then, the score of each match (A, B) according to our features becomes $\sum_i F_i(A, B)$.

Given that what we are trying to do is disambiguate between matches to the same function B in T, we would like to define a likelihood over all possible matches. Let us define B a function in T and $(A_j)_{j<n}$ the list of functions from S which have been structurally matched to B. We initialize our likelihood $(L_{B,A_j})_j$ uniformly to $\frac{1}{n}$. We then compute the score of all matches $S_{B,A_j} = \sum_i F_i(A_j, B)$. In order to normalize in a way that will give exponentially more importance to higher scores, we use the softmax function : $\bar{S}_{B,A_j} = \frac{e^{\sum_i F_i(A_j,B)}}{\sum_k e^{\sum_i F_i(A_k,B)}}$. Finally, we update the likelihood scores to reflect the findings of our features (we normalize it so that it will sum up to 1) : $L'_{B,A_j} = \frac{L_{B,A_j}\bar{S}_{B,A_j}}{\sum_k L_{B,A_k}\bar{S}_{B,A_k}}$.

The above defines a single iteration of the feature scoring process; as features may rely on other nodes' current likelihood, we make it possible to run multiple iterations to convergence. It should be noted that we also define a likelihood quantity for functions which are part of the substructure matched by "head" functions. For example, if B is matched to A and we have $A \to A'$ and $B \to B'$, then we define the likelihood that B' corresponds to A'. This is done similarly to the initialization of the likelihood defined above, except that it is not done uniformly but proportionally to the number of times a match occurs. For example, if B was also matched to C and D and we had $C \to A'$, $D \to D'$, then we would have $L_{B',A'} = \frac{2}{3}$ and $L_{B',D'} = \frac{1}{3}$. This likelihood value is not updated during the feature scoring process and is merely present so that it may be used by features.

B. Proposed features

While our features can be made to reflect the structure of the call-network, they can also work by comparing attributes of the functions being matched; these attributes can be associated to each node by the code crawler without performing any in depth analysis or actually comparing code across programs. With this in mind, we define a few features which we believe will help disambiguate matches based on previous oservations :

- In order to make use of the context in which functions are used for disambiguation, we define a feature based on matching sibbling functions across call-networks. We define a sibbling function to B as a function which has a parent in common with B. Sibbling sets can be obtained efficiently for each node with the right pre-processing; in our implementation, we simply pre-process the list of parent nodes for each node and then take the union of their sets of children. When scoring the match $(A \in S, B \in T)$, our feature computes S_A and S_B, the sets of sibblings for A and B respectively. We then define the score $F_0(A, B) = \sum_{s_B \in S_B} \sum_{s_A \in s_A} L_{s_B,s_A}$ the sum of likelihoods of all possible matches across the sets of siblings. In practice, most of these likelihoods will be 0, and this will amount to counting how many sibblings we think the functions have in common, each modulated by the likelihood that each match is correct. This should help formulate the idea that two functions are more likely to match one another if they are called in similar contexts.

- In order to discourage wildly improbable matches, we introduce a feature which compares the length of both functions being matched. This is based on the intuition that matching functions are likely to be similar in length. In order to do this, we have our crawler assign a function length attribute to each of our nodes, and we define our feature $F_1(A, B) = e^{-\frac{|l_A - l_B|}{c}}$, where l_A and l_B are the lengths of A and B respectively, while c is a constant we tune for our needs. The feature returns 1.0 for identical lengths and increasingly small scores the bigger the gap in length is.

- Another feature designed to exploit more of the call-network structure we added relies on comparing the relative number of paths from the root to A with B's,

the intuition being that the relative number of paths is an indicator of how often the function is used in the code. The formula for F_2 is the same as for F_1, simply using a different c constant. This feature is likely to be very useful when comparing an updated version of a binary with an older one, but whether it will have a positive impact in general remains to be seen.

There are other features which could benefit the matching process. For example, knowing how many arguments each function takes would be tremendously helpful for disambiguation. Unfortunately, reliably extracting that attribute would require complex code analysis which is outside the scope of this project.

V. Data collection

In order for this sytem to work, we also needed to develop a way of retrieving the call-network from a given binary. We arbitrarily chose to work with the ARM architecture for this due to personal preference; however, this same general approach should be applicable to any other instruction set. We developed a code "crawler", which is fed a compiled code binary and entrypoint, and then analyzes instructions in sequence the same way a processor would. When a branch-with-link instruction is detected, a node is created for the corresponding function, and the function-call edge is also created. A table is used to keep track of which instructions have already been visited. Whenever a conditional branching instruction is encountered, the crawler "forks" in order to pursue all possible paths. The crawler was also written with awareness of some common compiler optimizations when it comes to function calls, such as when an unconditional branch instruction is used at the end of a function instead of a branch-with-link instruction. A pseudocode version of the crawling algorithm can be found in Algorithm 2.

Algorithm 2 Algorithm for generating a call-network from ARM code. The algorithm presented here is incomplete and is only meant to give an idea of what is involved

```
function CRAWL(offset)
    loop
        if visited(offset) then
            return
        end if
        visited(offset) ← true
        inst ← get_instruction(offset)
        if inst.id == ARM_INS_BL then
            CREATE_EDGE(cur_func, inst.dest)
            CRAWL(inst.dest)
            offset ← offset +4
        else if inst.id == ARM_INS_B then
            if inst.conditional then
                CRAWL(inst.dest)
                offset ← offset +4
            else
                offset ← inst.dest
            end if
        else if inst.id == ARM_INS_BX then
            return
        else
            offset ← offset +4
        end if
    end loop
end function
```

VI. Results

A. Implementation

Both the crawler and the matching algorithm were implemented in python. The crawler was implemented with the help of capstone [5], a disassembly framework, in order to disassemble instructions. The matching program itself however was implemented without making use of any library, with everything being implemented from scratch, including the AHU tree isomorphism algorithm. Although time complexity was initially expected to potentially be problematic, the python implementation ended up being very fast, as the results in Table I show.

B. Evaluation metrics

In order to determine the quality of our algorithm's results on real-world programs, we need to define what quality means in this context. In each experiment, we will define a list L of functions from S which we want to find in T. We then define true positives as instances where a function from L is correctly matched to a function in T, false positives as instances where a function from L is incorrectly matched to a function in T, and false negatives as instances where a function from L exists in T but is not matched correctly. Given this, we are able to define canonical precision $= \frac{\text{true positives}}{\text{true positives}+\text{false positives}}$ and recall $= \frac{\text{true positives}}{\text{true positives}+\text{false negatives}}$. Given this, we can define an overall score using the F_1 measure defined as $F_1 = 2\frac{\text{precision.recall}}{\text{precision}+\text{recall}}$. These measures will be helpful to understand how and why our system performs the way it does. High precision scores indicate that few functions from L were assigned to the wrong node in T, while high recall scores indicate that most functions in L were found in T.

C. Experiments

We ran several experiments to evaluate our system's usefulness in various use cases. The numerical results for all of these experiments can be found in Table I.

The first experiment we ran used two binaries of the same program, 3ds_hb_menu [7], one built using a commit from several months ago and one built from the latest commit. This is meant to simulate on use case of our system, which is to port over notes from a given program to an updated version of that same program. In this case, we do not limit ourselves to trying to identify functions from a single library, but instead try to identify every function from the program. The results are very good, with an F_1 score of 89.3% and both precision and recall scores very close. This indicates

that our system performs very well in this use case, which is unsurprising given that all our disambiguation features would definitely work well in this context. For example, sibblings are unlikely to have changed very much, and the same is true for function lengths.

TABLE I
RESULTS OF EXPERIMENTS

S	T	Library	Precision	Recall	F_1	Time
hb_menu	update	All functions	92.0%	86.7%	89.3%	200 ms
hb_menu	portal	ctrulib / libc	51.5%	98.9%	67.7%	359 ms
portal	hb_menu	ctrulib / libc	64.2%	96.6%	77.1%	259 ms
scr_tool	hb_menu	ctrulib / libc	50.3%	92.3%	65.2%	121 ms
scr_tool	portal	ctrulib / libc	49.4%	98.7%	65.8%	291 ms
scr_tool	portal	lodepng	100.0%	88.9%	94.1%	293 ms
portal O0	portal O2	ctrulib / libc	75.9%	97.7%	85.4%	474 ms
hb_menu O2	portal O0	ctrulib / libc	40.2%	93.3%	56.2%	617 ms

Our second experiment consistend in trying to identify functions from specific libraries across unrelated code binaries. The libraries in question were the standard C library and ctrulib [9], a library designed to interface with an embedded operating system. We ran this experiments across 3 pairs of binaries which use these libraries, including portal3DS [6], 3ds_hb_menu and scr_tool [8]. We used these binaries because they are very variable in size and complexity. What we can see is that across all 3 experiments, the results are very similar : precision is low, around 50%, while recall is extremely high, always greater than 90%. This means that in these cases, while most functions we were looking for were successfully found, many other functions from T were incorrectly identified as being one of the ctrulib or libc functions we were trying to find. One of the main reasons for this is our system's design : while we put significant focus on disambiguatinig cases where multiple functions from L were matched to a single function from T (many-to-one ambiguities), our system does not attempt to disambiguate between the same function from L being matched to several functions from T (one-to-many ambiguities). That, coupled with the fact that ctrulib is a fairly "shallow" library, as can be seen in Figure 3, means that there is a high probability that ambiguities will happen in both directions. While clearly the high recall scores indicate that our feature based disambiguation is effective, it does not affect the other possible type of ambiguities. Interestingly, we also noticed that inverting S and T can change the results drastically : in the case of matching portal3DS and 3ds_hb_menu, swapping them resulted in the precision increasing significantly, leading to an F_1 score of 77.1%, up from 65.2%. This is likely due to the fact that 3ds_hb_menu is a smaller binary than portal3DS, which leads to fewer opportunities for false positives, which in turn increases precision.

In order to confirm the conjecture we made to explain the last experiment, we ran another experiment using another library : lodepng [10]. As can be seen in Figure 4, this library is not shallow like ctrulib and therefore should have enough internal structure for our system to successfully match

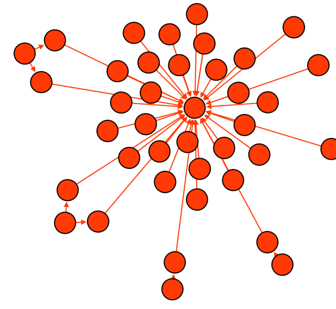

Fig. 3. Call-graph a subsection of ctrulib. Most functions only do a single function call and are not called by another ctrulib function. This makes it difficult to recognize functions based on network structure alone. .

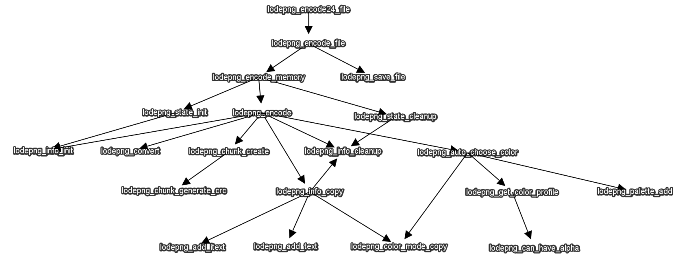

Fig. 4. Example call-graph for a lodepng function. There most functions make multiple calls and there is a lot of complex structure to work with. .

its functions across binaries. As we can see in the results, this seems to be the case : the experiment trying to find portal3DS's lodepng functions in scr_tool achieves perfect precision and high recall, resulting in an F_1 score of 94.1%. This seems to confirm that our system is capable of correctly matching library functions given that the libraries have more easily recognizable internal structures.

Finally, as one of our reasons for preferring call-structure comparison over code analysis is compiler variation, we performed experiments comparing binaries compiled using different optimization levels. We started by comparing the same program, portal3DS, compiled with GCC's O0 and O2 optimization levels. The matching process was quite successful in this case, with an F_1 score of 85.4%. However, this is not sufficient to evaluate our system on varying optimization levels, as using the same program as both S and T gives the advantage of having more reliable contextual information. Therefore, we ran an experiment matching functions from 3ds_hb_menu compiled with O2 to portal3DS compiled with O0. The results are that precision is very poor, falling to just 40% from the 51.5% when both programs use the same optimization level, while recall stays very high. This high recall score indicates that most structure is still present and that the difficulty in this case is mostly with dealing with one-to-many ambiguities. Overall, the result remains comparable to what it was using the same optimization levels, which is much better than what we could expect from a code

analysis based system. These differences in performance are expected : although ideally optimization should not affect our call-network, in practice compilers can sometimes modify the call-network by for example inlining certain functions, which in our context is like merging multiple nodes together. Fortunately, this sort of case is relatively rare, as is shown by our high-recall results.

VII. CONCLUSION

In this paper, we set out to determine whether relying entirely on call-network structure would be enough to match library function across compiled code binaries. To this end, we devised an algorithm which solves the subgraph isomoprhism problem efficiently in this specific context. Faced with the problem of function match ambiguities, we used a feature-based scoring system for decision making. We ran several experiments and determined that while our system can work perform very well in situations where either much of the structure is shared (such as in the case of updates of the same program) or when the target libraries are rich in complex topological structures, it is not able to perfectly solve the problem in the general case. We believe this is due to the fact that we do not currently have a good way to solve one-to-many ambiguities. Despite this, our system still performs well in the sense that it is typically able to match functions in T with the right function in S (high recall), even when faced with code variations due to different code generation tactics (such as those related to compile optimization flags), which is a case that a code analysis based system would be likely to perform poorly in. This seems to prove that call-network structure is in fact a rich source of information, and that while it may not be enough to solve the issue on its own, it would be a powerful way of augmenting existing solutions such as IDA Pro's FLIRT [4].

REFERENCES

[1] Stephen A. Cook, *The Complexity of Theorem-Proving Procedures* 1971.
[2] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento, *A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs*.
[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms* 1974.
[4] *IDA F.L.I.R.T. Technology: In-Depth* https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml
[5] *Capstone - The Ultimate Disassembly Framework*. http://www.capstone-engine.org/
[6] *portal3DS*. https://github.com/smealum/portal3DS
[7] *3ds_hb_menu*. https://github.com/smealum/3ds_hb_menu
[8] *scr_tool*. https://github.com/smealum/scr_tool
[9] *ctrulib*. https://github.com/smealum/ctrulib
[10] *lodepng*. http://lodev.org/lodepng/