RESource: A Framework for Online Matching of Assembly with Open Source Code

Ashkan Rahimian†, Philippe Charland‡, Stere Preda†, and Mourad Debbabi†

† Computer Security Laboratory, CIISE
Concordia University, Montreal, Quebec, Canada
{a_rahimi,s_preda,debbabi}@encs.concordia.ca
‡ Mission Critical Cyber Security Section,
Defence R&D Canada - Valcartier, Quebec, Canada
philippe.charland@drdc-rddc.gc.ca

Abstract. Software reverse engineering is a fastidious task demanding a strong expertise in assembly coding. Various existing tools may help analyze the functionality of a binary file without executing it and an interesting step would naturally be the search for the original source files. Our tool called RESource considers the extraction of some features in the assembly code so that queries can be triggered to a source repository in a reliable way: either (1) the result is a set of references to the original project files provided they are hosted on the repository or (2) at least some functionalities of the binary file are unleashed. Such an approach is very promising given its proved performances in real assembly code applications.
Keywords: reverse engineering; assembly code; source repository.

1 Introduction

Software reverse engineering consists in studying and understanding the process by which a machine-generated assembly language program has been created by working backward [3]. If manually writing assembly ASM code involves specific programming skills, a compiler automatically converts a high-level language such as C into machine code. The ASM analysis becomes extremely challenging, especially if the compiler adds certain optimizations by rearranging the computations, changing or replacing some operations.

Common reverse engineering practices suggest two approaches – dynamic and static – with the binary file as the starting point. By dynamic approach (e.g., [11]) we mean isolating the binary file in an application specific environment to model its behavior by execution. Since this does not necessarily reveal all execution flows, debugging tools (e.g., WinDbg [18], Gdb [5], Valgrind [17]) are often associated with this method. As long as only the functionality is targeted, the dynamic approach is acceptable. In other situations, static analysis yields better results and does not compromise the security requirements of the analysis environment.

The first step of the static analysis of a binary file is the disassembly phase. The disassembler (e.g., objdump [6] in Linux) is a program considered of invaluable help since it generates the ASM code of the binary file. At this level, mastering the ASM program representation seldom leads to fully understanding the program functionalities. More advanced disassemblers such as IDA Pro [9] are meant to simplify the analysis by offering a rich GUI with the program divided into basic blocks in a program flow graph (PFG). A challenging further step is then to obtain a correct higher-level program representation, i.e., the source files. A decompiler (e.g., Hex-rays [14] or TyDec [16]) could help a lot but since there is not always a 1:1 correlation between the ASM and the source objects, the automatically generated sources may be difficult to follow. For example, it is not simple to detect the definition of object structures in ASM.

Our current purpose is a tool – RESource – to help enforce the mapping between the machine and the source code. For that we draw inspiration from the RE-Google project [10]. RE-Google was designed on top of the GData framework and Google Code Search APIs [7] which were officially deprecated. Our tool provides a functionality similar to RE-Google and introduces new ones. The approach is the following: with the principle of code reuse in mind, our tool will exploit some features that exist at both the source and the assembly file levels. The tool is thus able to trigger queries based on these features on certain repositories used by the developers' community. If there are no query results, the tool is still able to give us some information about the functionality of a portion of the ASM file using an Offline Analyzer module. The information returned by this module are related to the function stack frame, prototype, arguments, local variables and low-level system calls. It has a built-in dictionary of common user and kernel level API functions that are used by malware to interact with the Windows operating system for performing tasks such as file I/O, network communications, registry modification, working with services, etc.

The reminder of the paper is structured as follows: first we shall introduce the motivation of our work and related background. In Section 3 we present our methodology followed by implementation details. Two experimental scenarios are described in Section 4, followed by the conclusion in Section 5.

2 Motivation and related work

The current work pertains to the domain of static analysis of ASM code and more precisely, in the mapping of ASM to source files. Though a decompiler seems to be the program which best fits our goals, we consider that an attempt of mapping the ASM to existing source code should come first. Applications such as malware analysis can grasp the benefits of a tool able to give reliable information about the standard and open source files used by a malicious developer. Decompilers, methods and tools to analyze malware code already exist. They can be used by expert reverse engineers who seek to understand the origins and the creation process of the malware.

IDA Pro allows disassembling a binary file and its rich GUI simplifies the analysis of the ASM code. It is widely used thanks to its multiple features, such as the possibility of integrating a debugger like WinDbg (which became the de facto Microsoft debugger since Softice stopped being maintained) and more interesting, plugins such as the Hex-Rays decompiler - "the most advanced decompiler ever built!" [14]. Although there have been a few attempts to design generic debuggers to work on heterogeneous platforms (e.g., GenDbg [4]), IDA Pro proves to be one of the most complete tool in reverse engineering.

The ASM code follows rather regular patterns. Consequently, the decompiler is able to do a mapping between registers or memory locations, abstract variables, and thus extract for example, a C-like program from the ASM file (indeed, most of the decompilers are not generic). Other basic C constructs (e.g., loops) are more difficult to extract and some decompilers fail to solve them (e.g., Boomerang [2]). Another challenging problem is reconstructing the abstract types (e.g., structures). TyDec [15] tries to tackle the problem but is limited to an experimental level. In this case, the best practice remains the human expertise, i.e., the definition of a structure which is guessed after a first decompilation is manually introduced and the C program is then rewritten.

Our approach is rather different in that our RESource tool is meant to inform the reverse engineer about the standard and open source components that might have been used by the creator of the binary file. To our knowledge, a similar functionality is ensured only by the IDA Pro RE-Google plugin [10].

RE-Google, written in Python, relies on the IDA API and the Google Code Search API [7]. It takes the disassembled binary code as input and creates a query submitted to Google Code Search based on the constants, strings, and function names. The response from the search engine is the potential source excerpt that contains similar code. Although it supports a limited set of features to create a query, RE-Google may confine the search to certain languages. Additionally, it can be configured to search for a specific function within the disassembly, skip certain functions, or perform a search for all available functions. Also, the interval between two subsequent searches can be defined. Optionally, user credentials could be supplied as part of the query to the code search engine. Furthermore, there is an option for restricting (blacklisting) certain string patterns in the result. Similarly, a constant filter function checks the immediate values and removes flags and small values form the query if they are not relevant for the search. The response from the search engine is parsed and the top results are added to the code as comments.

Our goal is to design RESource as an IDA Pro plugin too, making use of code search engines for open source software. In addition, we want to have the capability to search in newsgroups and user-defined code repositories, taking thus into account a larger panel of search engines than RE-Google. RESource does not only provide extended queries by adding new features. It also allows to reveal parts of the code functionalities whenever the query results are null. In the next section, we describe the methodology and the concrete implementation details of our RESource tool.

3 Methodology

The input to our process is the ASM file resulting from the disassembly of a target binary in IDA Pro. The specific representation of the ASM, together with its PFG, lead us to consider the partitioning of the ASM code in blocks, each one corresponding to begin *proc / end proc*, where proc stands for procedure. Here is an example of a simple code in C.

```
int sum(int a, int b){
     return a + b;
}
```

The corresponding ASM code contains a procedure that we can easily identify by its *name* "sum" (IDA Pro encloses it with the *begin proc* and *end proc* keywords). **sum** :

```
push   %ebp
mov    %esp,%ebp
mov    0xc(%ebp),%eax
add    0x8(%ebp),%eax
pop    %ebp
ret
```

We thus consider an ASM file as a set of procedures that are to be individually analyzed by our tool. Each procedure may contain some *interesting features* (see Section 3.2) that our tool is able to extract and exploit in order to submit queries to a source repository. The result is (1) either a set of links to pertaining source files referencing the same features and which are inserted as comments in the original ASM file or (2) the insertion of a comment about the functionality of the current procedure after its local offline analysis (cf. Section 3.2).

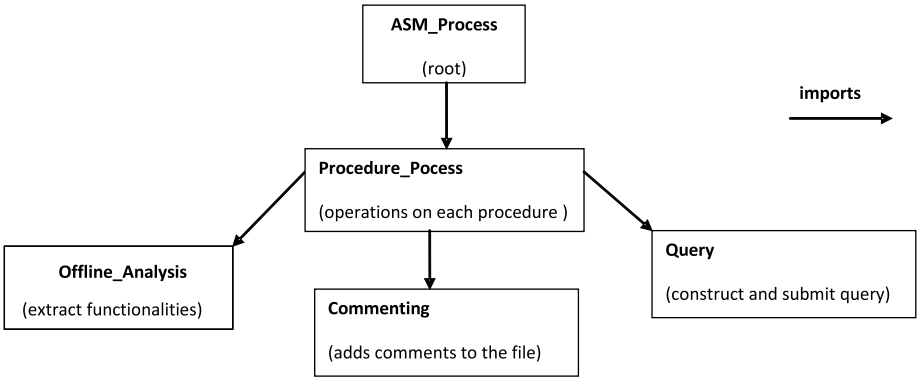


Fig. 1. Algorithm Decomposition.

3.1 Algorithm

We adopt a B-Method like notation [1] to describe the algorithm implemented by our RESource tool. Fig. 1 captures the RESource algorithm decomposition in B-like components. The algorithm has five modules: (1) *ASM_Process* root machine which provides the interface with the user. It imports the (2) *Procedure_Process* machine responsible for processing each ASM procedure. It calls the operations of the (3) *Query* module in order to submit queries to a *set* of code repositories. The (4) *Offline_Analysis* module is in charge of a local analysis to extract the program functionality and also the operations of the (5) *Commenting* module which adds the pertaining comments to the original file.

***ASM_Process* module**

```
MACHINE ASM_Process
IMPORTS Procedure_Process
SETS
    PROCEDURES
CONSTANTS
    ASM_Original_file
PROPERTIES
    ASM_Original_file ∈ ℙ(PROCEDURES)
VARIABLES
    Some_Procedures
INVARIANTS
    Some_Procedures ⊆ ASM_Original_file
INITIALISATION
     Some_Procedures := ∅
OPERATIONS
    try_read_procedures(procs) = PRE procs ≠ ∅ ∧ procs ⊆ ASM_Original_file
                                    THEN Some_Procedures := procs
                                 END;
     process = PRE Some_Procedures ≠ ∅ THEN
                 VAR F1, F2, p IN
                    WHILE Some_Procedures ≠ ∅ DO
                        ANY p WHERE p ∈ Some_Procedures THEN
                          F1, F2 ← read_features(p); /*from Procedure_Process*/
                          query(F1); /*op. in Procedure_Process*/
                          analyse_locally(F2); /*op. in Procedure_Process*/
                          update(p); /*op. in Procedure_Process*/
                          Some_Procedures := Some_Procedures \ {p};
                        END
                    END
                 END
                END
END/*ASM_Process*/
```

Fig. 2. ASM_Process module.

Any ASM file is a SET of PROCEDURES. As we can easily depict from Fig. 2, we take as input to our algorithm the ASM_Original_file. It is of type PROCEDURES and remains CONSTANT. These assumptions are captured by the CONSTANT and PROPERTIES clauses. The only variable we introduce is a set of SOME_PROCEDURES among those presented by IDA Pro that the user chooses to analyze. This variable may be modified by the OPERATIONS which must always satisfy the INVARIANT. Here, the INVARIANT states that the procedures to be analyzed are part of the original ASM file.

Procedure_Process **module**

MACHINE *Procedure_Process*
IMPORTS *Query, Offline_Analysis, Commenting*
SETS
FEATURES
VARIABLES
OnFeat, OffFeat, queried, analysed, updated
INVARIANTS
OnFeat ⊆ FEATURES ∧ OffFeat ⊆ FEATURES ∧ queried ∈ **BOOL** ∧
analysed ∈ **BOOL** ∧ updated ∈ **BOOL**
INITIALISATION
OnFeat, OffFeat, queried, analysed, updated := ∅, ∅, false, false, false
OPERATIONS
F1, F2 ← *read_features*(p) = **PRE** p ≠ ∅ **THEN**
`/*features extraction from p : to refine*/`
F1:=OnFeat; `/*features for online analysis*/`
F2:=OffFeat; `/*features for local analysis*/`
queried, analysed, updated := false, false, false;
END;
query(f) = **PRE** f ⊆ OnFeat ∧ queried = false **THEN**
IF f ≠ ∅ **THEN**
submit_query(f); `/*operation in Query machine*/`
END
queried := true;
END;
analyse_locally(f) = **PRE** f ⊆ OffFeat ∧ queried = true ∧ analysed = false
THEN `/*local analysis for functionality extraction */`
`/*based on operations in Offline_Analysis machine*/`
analysed := true; append_to_log(f); `/*displaying results*/`
END;
update(p) = **PRE** queried = true ∧ analysed = true ∧ updated = false **THEN**
`/*updates after the online query and the local analysis*/`
`/*based on operations in Commenting machine*/`
updated := true;
END;
END`/*Procedure_Process*/`

Fig. 3. Procedure_Process module.

For each procedure, there is a phase of ASM *features* extraction, followed by the submission of queries to source repositories and a local analysis.

We explain these steps in the *process* operation of Fig. 2. The Procedure_Process module uses respectively the services of the Query and the Offline_Analysis modules for the specific *query* and *analyse_locally* operations (Fig. 3). These operations are to be carefully implemented since their abstract representation cannot contain too many details. The module states only the permitted order in which these operations are called via the PRE-condition clause.

The *process* operation of Fig. 2 considers a last phase of *updating*. The original ASM file remains the same, i.e., *constant*, except for the ASM comments part which gathers the query results and the local analysis. Therefore, the result of the entire process is the original file updated with comments as we shall see in Section 3.2.

Query module

MACHINE *Query*
CONSTANTS
n, SEQ_REPS
DEFINITIONS
Repositories == 1..n
PROPERTIES
n ∈ ℕAT1 ∧ SEQ_REPS ∈ Repositories → Repositories
VARIABLES
Queryable_Reps
INVARIANTS
Queryable_Reps ∈ Repositories → **BOOL**
INITIALISATION
ran(Queryables_Reps) := true
`/* all repositories should be queryable at the beginning*/`
OPERATIONS
submit_query(F) = **ANY** r **WHERE** Queryable_Reps(r) = true **THEN**
`/*submit query*/`
Queryable_Reps(SEQ_REPS(r)) := true;
END
END`/*Query*/`

Fig. 4. Query module.

Based on the extracted features in a procedure, the role of the Query module is to construct and submit queries to a set of source repositories which are previously known. We could use an instantiated SET of repositories to capture this information, but for the sake of simplicity, we choose to identify each source repository with a natural number in the set 1..n, where n is the number of repositories.

Moreover, we also express the following requirement: a real source repository may not be queried too frequently (e.g., wait a few seconds between each query). Consequently there should be a mechanism to launch the query to a different *queryable* repository so that the process does not stop. The straightforward way is to introduce a CONSTANT function SEQ_REPS which gives the *next* source repository to query. Implementing this is based on the observation of some query interval slots for each real repository and by thus defining an order of passing from one repository to another. Queryable_Reps(r) = true therefore means that the r repository can accept a query. This variable is modified in the implementation of the submit_query operation.

We do not give the B notation of the Offline_Analysis and Commenting modules because their operations proved to be more challenging to implement at low level. The *append_to_log*() operation is meant to save the execution steps in a log file at runtime.

3.2 Implementation Details

If a B-like algorithm description is useful to examine the possible flows and to define the operations preconditions and the invariants they have to meet, the validity of the low level implementation is generally asserted using normal techniques such as testing and peer code reviewing.

RESource program implements the algorithm as a Python IDA Pro plug-in. It is worth mentioning that, unlike the RE-Google plugin [10], our extended version does not rely on the GData framework [8], nor does it utilize Google Code [7] as the only search engine for accessing code repositories. Instead, it possesses a built-in query processing engine and parsing mechanism for handling request/response messages. Furthermore, it supports multiple search engines and it provides a framework for adding new code repositories with only a few lines of code. Also, the program makes use of an interleaving time optimization technique for managing multiple search engines. Despite the large number of request/response messages, it honors the required time delays between consequent messages without wasting processing time.

In terms of extracted *interesting features* from the ASM code, RESource is able to get four types of features for online analysis and query building: (1) immediate values of operands, (2) imported libraries and function calls, (3) exported functions in DLLs, and (4) strings values. In addition, it considers eight features for offline analysis. For each function, we extract information about its stack frame: (1) number of instructions; (2) size and number of local variables; (3) size and number of arguments; (4) size of saved registers; (5) function flags; (6) function addresses (begin, end, return); (7) function prototype (type of input and output and calling convention); (8) calls to low level system functions (malware dictionary). Moreover, variable scopes (local/stack-based or global/memory-based) and simple data structures (single variables or *structs*) are also highlighted for the reverse engineer.

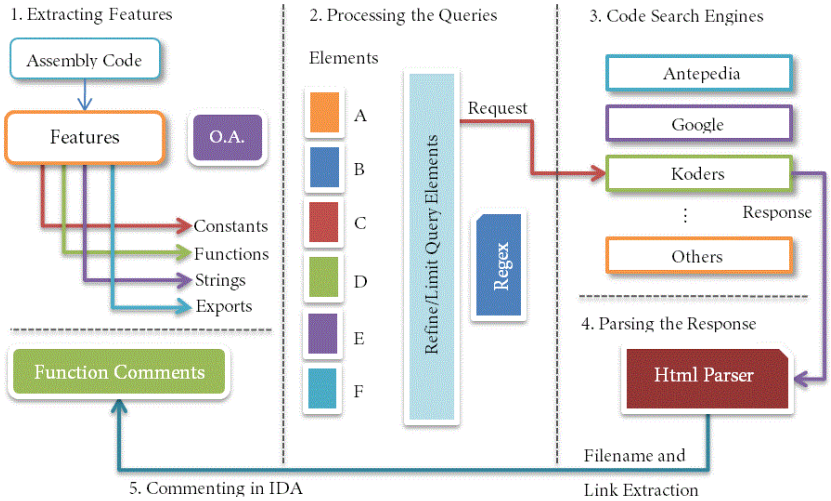


Fig. 5. Execution Flow

Moreover, the program adds better result handling techniques than RE-Google and an offline functionality analysis engine. In many situations, online results may not be available due to the lack of extracted features, obfuscated or hard-coded procedure, use of complex and non-standard algorithms, etc. Therefore, the offline analyzer is of great benefit for revealing the overall functionality of a portion of assembly code. It has an extendable dictionary of common functions in Windows API along with a programmer-friendly description of each function.

Execution Flow

As illustrated in Fig. 5, there are five main modules in the Python program for handling tasks related to Features, Queries, Repositories, Parsing and Commenting. Except for the Code Search Engine (3), these modules have a counterpart in the algorithm blocks of Fig. 1. The RESource program interacts with the IDA Pro API for getting a list of available procedures in the disassembly, getting function addresses and names, as well as adding comments to the file.

The execution flow starts in the main function of the script where the initialization of variables and execution time calculation is done ($Initialize(RESrc\_Vars)$). Then, the script checks a variable (flag) to determine whether the search should be performed on a specific function or on all the extracted functions from the disassembly ($RESrc(asm\_function\_list)$). In the first case, the user highlights a specific function for search and in the second case, all the functions are taken into account.

In the next step, the RESrc function counts the total number of available procedures and prepares a loop for analyzing each item. Then, a function will be

called for extracting four types of features, namely constants, imported libraries, exported libraries and string values from the disassembly. The output of the Extract_QFeatures() function is a potential list of features that could be used for building a general query. This list will be refined several times before building a specific query. Next, the features for offline analysis are extracted using the Extract_OAFeatures() function :

```
feature_qlist ← Extract_QFeatures( asm_function )
feature_oalist ← Extract_OAFeatures( asm_function)
            // OA for Offline Analysis
    feature_list_func_i = [ f_i1, f_i2, ..., f_in ]
refined_qlist ← Refine_GQuery( feature_qlist )
```

The purpose of the *Offline Analysis* (OA) module is to compare a function with a list of known Windows API functions in order to get a simple statement about the functionality and prototype of the procedure under analysis. Also, this module assists the reverse engineer by highlighting the variables and their scope.

The purpose of the Refine_GQuery() function in the refining process is to filter out certain characters from the feature list to prevent problems with search engines queries. For instance, the search engines may not allow characters such as "%, ', ' "' as part of query string to prevent SQL injection. Therefore, the output query is safe for submission into code search engines. However, the user can define what characters are blacklisted by adding 'badkey': 'value' pairs into the "BlackDict" dictionary. For instance, the keys in the following dictionary are simply replaced with the '' character which is equivalent to removing them from the search string.

```
BlackDict = {'%d':'', '%s':'', '\\':'', '%1':'', '%2':'', ...}
```

Also, this function encodes and prepares the list for the next steps of specific query building:

```
base_query ← Generate_Query( refined_qlist )
```

At this step, we have a base query that can be further encoded for particular search engines. Search engine-specific prefix and suffix will be added to each base query to build a standard query. The following functions are examples of query building functions for three major code search engines.

```
final_queryKoders ← Build_Koders_Query( base_query )
   final_queryGCS ← Build_GCS_Query( base_query )
final_queryKrugle ← Build_Krugle_Query( base_query )
```

The next step is to submit the query and get the response for each respective search engine. The order of query submission and response extraction is important for time optimization. Usually there must be a time delay between two

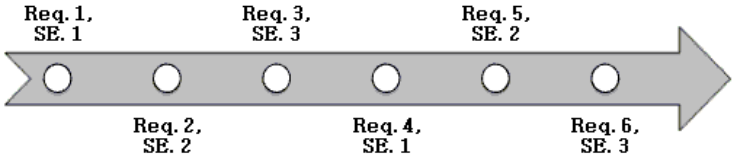


Fig. 6. Timing Interleaving.

subsequent requests to a search engine (SE). The program uses an interleaving technique for managing the query submission and for saving processing time (as shown for example in Fig. 6).

For each query, a request is made and the response page is received in HTML.

$$\texttt{html\_page}_j \leftarrow \texttt{Fetch\_Response(final\_query}_j \texttt{)}$$

After getting the page, a call to the local parsing function will be made to extract relevant information based on a predefined regular expression statement for each search engine. Then, the matching Filenames and URLs are extracted and stored in a dictionary.

$$\texttt{dictionary\_list} \leftarrow \texttt{Refine\_Results(Parse\_Page(html\_page))}$$

The results are processed and duplicate results are removed from the list. Also, based on the search engine rankings, the best matches are selected and given a priority. Lastly, the comments are updated to reflect the online search results.

$$\texttt{function\_comment} \leftarrow \texttt{Update\_Comment}_{function\_k}\texttt{(refined\_dictionary\_list)}$$

In the next section, we present a practical application of our RESource program on an open source software.

4 Experimental Results

We have adopted the PreciseCalc Project [12] given that both the sources and binary files are available on SourceForge and Koders (*http://www.koders.com*) as a code search engine. As an input to our RESource IDA Pro plugin, we use the ASM file resulting from disassembling the PreciseCalc binary. There are 533 ASM procedures and we choose to analyse all of them.

The `Extract_QFeatures()` function is able to extract features from 67 procedures. If there are at least two elements in the *Imports list* or the joint set of *Constants* and *String List* is non-empty, then the script would try to find an *exact match* by concatenating all the elements. This is an ideal situation where the

1	* Analyzing function 4 [sub_4013A0] @ [0x4013a0] Constants: ['0x13880', '0xf4240', '0x4c4b40', '0x493e0'] Strings: [] Imports: set([]) list: 0x13880 0xf4240 0x4c4b40 0x493e0 * Looking for an exact match * * Looking for a rough match based on constants *
2	* Analyzing function 127 [sub_407F40] @ [0x407f40] Constants: [] Strings: ['"arctan, arccot from 1i or -1i"', '"argtanh,argcoth from 1 or -1"'] Imports: set([]) list: "arctan, arccot from 1i or -1i" "argtanh,argcoth from 1 or -1" * Looking for an exact match * * Looking for a close match based on strings *
3	* Analyzing function 334 [sub_417B20] @ [0x417b20] Constants: ['0x80000001', '0x80000001', '0x80000001'] Strings: ['"Software\\Petr Lastovicka"', '"Software\\Petr Lastovicka"'] Imports: set(['RegDeleteKey', 'RegOpenKey', 'RegCloseKey', 'RegQueryInfoKey']) list: 0x80000001 0x80000001 0x80000001 "Software\Petr Lastovicka" "Software\Petr Lastovicka" RegDeleteKey RegOpenKey RegCloseKey RegQueryInfoKey * Looking for an exact match * * Looking for a close match based on strings * * Looking for a rough match based on constants *
4	* Analyzing function 375 [sub_41B4A0] @ [0x41b4a0] Constants: [] Strings: ['"sin"', '"cos"', '"tan"', '"tg"', '"cot"', '"cotg"', '"asin"', '"acos"', '"atan"', '"atg"', '"acot"', '"acotg"', '"arcsin"', '"arccos"', '"arctan"', '"arctg"', '"arccot"', '"arccotg"', '"arc"'] Imports: set(['strnicmp']) list: "sin" "cos" "tan" "tg" "cot" "cotg" "asin" "acos" "atan" "atg" "acot" "acotg" "arcsin" "arccos" "arctan" "arctg" "arccot" "arccotg" "arc" strnicmp * Looking for an exact match * * Looking for a close match based on strings *

Fig. 7. Feature Extraction (excerpt from the *log* file).

query would be expressive enough in terms of number and the type of elements. If no exact match is found, then the search would be based on the strings inside the binary. If the length of *String List* is larger than one, then the search query will be built by concatenating the String elements. Finally, if there is at least one element in the *Constants List* but the results set is empty, the script will perform the search by building a query based on the concatenation of constant elements.

The conditional rules for defining each case can be altered based on the application under analysis and the number of available elements in the extracted lists. Generally, there are more elements in each list when the application makes use of Standard Windows Libraries.

Fig. 7 shows a few examples of interesting features extracted by the `Extract_QFeatures()` function. In the first one, the search is merely performed based on the *constants*. Example 2 shows a situation in which only *string* information is available. No import lists are detected for the first two cases. Conversely, in examples 3 and 4, sets of *imported function* names are included in the search. The original PreciseCalc project can be accurately identified by submitting a query containing portions of the strings in example 3. Even if an exact match is not found, the RESource program will try to find a close or a rough match based on a combination of features.

Table 1. Identified Source Codes.

Func. no.	Function ID @ Address	Source Code Reference	Match
70	[sub_406800] @ [0x406800]	complex.cpp	100%
146	[sub_409620] @ [0x409620]	lang.cpp	100%
159	[sub_40A1E0] @ [0x40a1e0]	matrix.cpp	100%
261	[sub_4119B0] @ [0x4119b0]	parser.cpp	100%
334	[sub_417B20] @ [0x417b20]	preccalc.cpp	100%

RESource was able to detect several references to each source file in the original project. PreciseCalc application includes functions that handle Text Editing, GUI Processing, Timing and Registry Modification, alongside the Arithmetic, Statistical, Geometrical and other math-related functions.

Table 1 shows sample results of the identified C++ source code. The identified links and filenames are inserted directly in the assembly file. There are several references to the main "preccalc.cpp" file. For instance, the functions at addresses 0x417b20, 0x41b2d0, 0x4190c0, 0x419ab0, 0x41a2b0, 0x41b4a0, 0x41be70 and 0x41c1f0 were referencing the main C++ file in the project. Fig. 8 shows one of these references.

RESource has detected several math-related functions in the disassembly. The script has generated a comprehensive execution log that is self-explanatory and describes the analysis process. Even though the current version of RESource does not include heuristic query processing techniques, it is able to detect more than 70% of the original source files with an accuracy of 100%. Also, the script is useful for gaining insight into the functionality of the ASM file.

Concerning the Offline Analysis module, the current implementation includes a dictionary of common Windows APIs alongside with a brief description of each one. This dictionary was built with malware analysis in mind. Therefore, it includes about 200 of the most common kernel and user level functions known to be used by existing malware [13].

```
.text:0041B2D0 ; =============== S U B R O U T I N E =======================================
.text:0041B2D0
.text:0041B2D0 ; Online _ preccalc.cpp _ http://www.koders.com/cpp/fidC00F18FC54602A45811CF096F055D2
.text:0041B2D0 ; Attributes: bp-based frame
.text:0041B2D0
.text:0041B2D0 sub_41B2D0      proc near               ; CODE XREF: sub_4099D0+55↑p
.text:0041B2D0                                         ; WinMain(x,x,x,x)+312↓p
.text:0041B2D0                 push    ebp
.text:0041B2D1                 mov     ebp, esp
.text:0041B2D3                 call    sub_418EE0
.text:0041B2D8                 call    sub_4190C0
.text:0041B2DD                 call    sub_4188D0
.text:0041B2E2                 push    offset aPreciseCalcula
.text:0041B2E7                 push    1F8h
.text:0041B2EC                 call    sub_409440
.text:0041B2F1                 add     esp, 8
.text:0041B2F4                 mov     off_425710, eax
.text:0041B2F9                 mov     eax, off_425710
.text:0041B2FE                 push    eax
```

Fig. 8. Identified "precalc.cpp" file @ 0x41B2D0.

In our second scenario, we run RESource on several malware disassemblies. The Offline analyser helps the reverse engineers to understand network connectivity and data gathering functionalities of malware by adding relevant comments.

In Fig. 9 there are several routines of malware performing file I/O operations and communication with a remote command and control server. In cases where RESource returns results from both the online code repositories and the Offline analyzer, an emergent consistency is observed. As an example, Fig. 9(e) depicts a portion of assembly code that is capturing the screen and saves it to a file to be remotely transmitted. As can be seen, RESource gives reliable information in both Offline and Online comment sections. Such rich informal expression of a comment may really be beneficial for the hectic job of a reverse engineer.

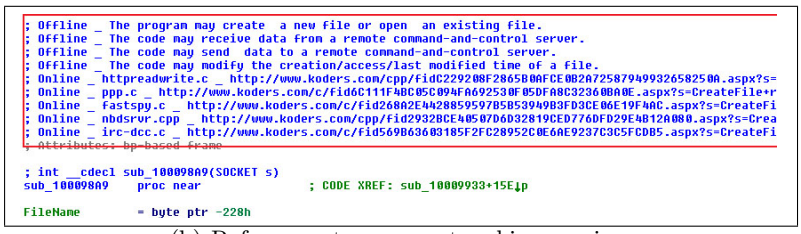
Discussion
A side by side comparison between the outputs of RE-Google and RESource was not possible because the underlying search framework of RE-Google was deprecated. In other words, RE-Google is not functional anymore. RESource takes an intra-procedural approach to extract features and build queries. It could be argued that an inter-procedural approach could improve the accuracy of the Online analysis. However, the search engines provide limited commands for executing logic-based queries and some of them do not provide direct APIs to their repositories. Adopting a heuristic query building algorithm that tries different elements in the query string and selects the best match could improve the accuracy of the identified online projects. As to the accuracy of the Offline analysis, it clearly depends on the number and selection of the functions defined in the dictionary. In a situation where we have results from both the online and offline analyzers, the reverse engineer would have the maximum information. This happens when programs make use of standard libraries such as VCL or MFC. In other cases, there might be no results from the online module. This happens when malware authors use non-standard components or they use certain wrappers around standard system calls. Also, they might use non-standard low level kernel functions for performing simple I/O operations.

5 Conclusions

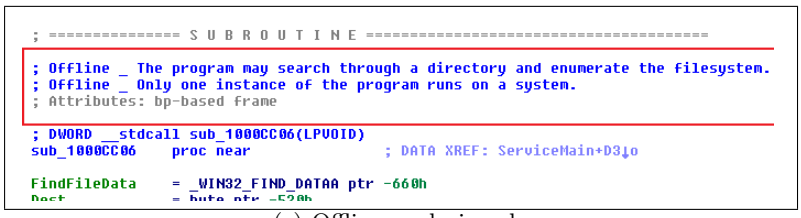
Software reverse engineering is a complex task. Applications like malware analysis can grasp the benefits of a tool able to automatically give reliable information about the matching between open source and assembly code.

In this paper, we established a framework to develop such a tool – RESource – that exploits some features existing at both the source and the assembly file levels. Based on these features, queries are triggered on certain online repositories used by the developers' community. If there is no query result, the tool is still able to provide some information about the functionality of a portion of the ASM file by a local offline analysis. The reverse engineer's task is thus greatly simplified.

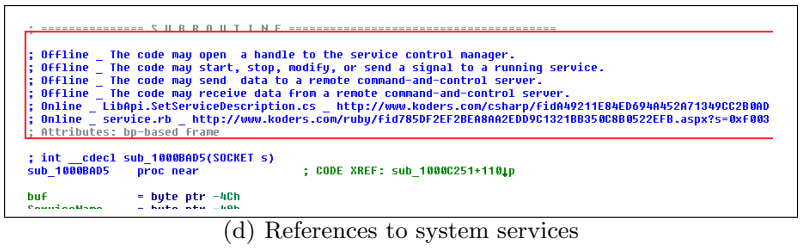
(a) Routine involving file I/O



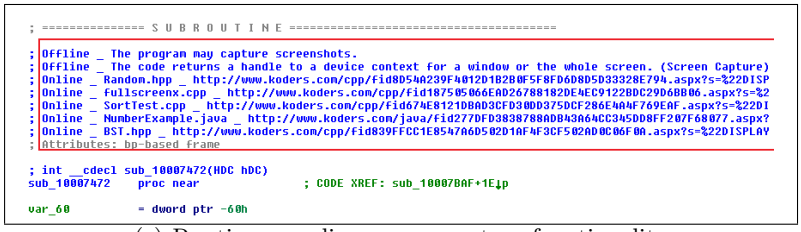
(b) References to some networking services



(c) Offline analysis only



(d) References to system services



(e) Routine revealing screen capture functionality

Fig. 9. Examples of the final outcome.

References

1. Abrial, J. R.: *The B Book - Assigning Programs to Meanings*. Cambridge University Press, ISBN 052149619-5 (1996).
2. Boomerang: a general, open source, retargetable decompiler of machine code programs. [On-line]. http://boomerang.sourceforge.net/.
3. Bryant, R. E., O'Hallaron, D. R.: *Computer Systems – A programmer's Perspective, 2^{nd} Edition*. Addison Wesley, ISBN 0136108040 (2010).
4. Eymery, D., Eymery, O., Borello, J-M., Fraygefond, J-M., Bion, P.: *GenDbg : un débogueur générique*. In: *Symposium sur la sécurité des technologies de l'information et des communications SSTIC'08*, France (2008).
5. GDB: The GNU Project Debugger. [On-line]. http://www.gnu.org/software/gdb/documentation/.
6. GNU Binutils. [On-line]. http://www.gnu.org/software/binutils/.
7. Google Code. [On-line]. http://code.google.com/.
8. Google Data APIs. [On-line]. http://code.google.com/p/gdata-objectivec-client/.
9. IDA Pro multi-processor disassembler and debugger. [On-line]. http://www.hex-rays.com/products/ida/index.shtml.
10. IDA Pro Re-Google Plugin. [On-line]. http://regoogle.carnivore.it/.
11. Lagadec, P.: *Dynamic Malware Analysis for Dummies*. In: *Symposium sur la sécurité des technologies de l'information et des communications SSTIC'08*, France (2008).
12. Precise Calculator Project. [On-line]. http://sourceforge.net/projects/preccalc/.
13. Sikorski, M., Honig, A.: *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, ISBN 1593272901 (2012).
14. The Hex-Rays Decompiler. [On-line]. http://www.hex-rays.com/.
15. Troshina, K., Chernov, A., Derevenets, Y.: *C Decompilation: Is It Possible?*. In: *Proceedings of International Workshop on Program Understanding*, pp. 18–27, Altai Mountains, Russia (2009).
16. Troshina, K., Derevenets, Y., Chernov, A.: *Reconstruction of Composite Types for Decompilation*. In: *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM '10*, pp. 179–188, Timisoara, Romania (2010).
17. Valgrind – a suite of tools for debugging and profiling. [On-line]. http://valgrind.org/.
18. WinDbg debugger for Microsoft Windows. [On-line]. http://www.windbg.org/.