IDA F.L.I.R.T. Technology: In-Depth

The Goal

One major stumbling block in the disassembly of programs written in modern high level languages is the time required to isolate library functions. This time may be considered lost because it does not bring us new knowledge : it is only a mandatory step that allows further analysis of the program and its meaningful algorithms. Unfortunately, this process has to be repeated for each and every new disassembly.

Sometimes, the knowledge of the class of a library function can considerably ease the analysis of a program. This knowledge might be extremely helpful in discarding useless information. For example, a C++ function that works with streams usually has nothing to do with the main algorithm of a program.

It is interesting to note that every high level language program uses a great number of standard library functions, sometimes even up to 95% of all the functions called are standard functions. For one well known compiler, the "Hello, world!" program contains:

```
        library functions       -       58
        function main()         -       1
```

Of course, this is an artificial example but our analysis has shown that real life programs contain, on average, 50% of library functions. This is why the user of a disassembler is forced to waste more than half of his time isolating those library functions. The analysis of an unknown program resembles the resolution of a gigantic crossword puzzle : the more letters we know, the easier it is to guess the next word. During a disassembly, more comments and meaningful names in a function means a faster understanding of its purpose. Widespread use of standard libraries such as OWL, MFC and others increase even more the contribution of the standard functions in the target program.

A middle sized program for Win32, written in C++, using modern technologies (e.g., AppExpert or a similar wizard) calls anything from 1000 to 2500 library functions.

To assist IDA users we attempted to create an algorithm to recognize the standard library functions. We wanted to achieve a practical, usable result and therefore accepted some **limitations**

- we only consider programs written in C/C++
- we do not attempt to achieve perfect function recognition : this is theoretically impossible. Moreover the recognition of some functions may lead to undesirable consequences. For example, the recognition of the following function

```
                push    bp
                mov     bp, sp
                xor     ax, ax
                pop     bp
                ret
```

  would lead to many misidentifications. It is worth noting that in modern C++ libraries one can find a lot of functions that are absolutely identical byte-to-byte but have different names.

- we only recognize and identify functions located in the code segment, we ignore the data segment.
- when a function has been sucessfully identified, we assign it a name and an eventual comment. We do not aim to provide information about the function arguments or about the behaviour of the function.

and we imposed the following **constraints** upon ourselves

- we try to avoid false positives completely. We consider that a false positive (a function wrongly identified) is worse than a false negative (a function not identified). Ideally, there should be no false positive at all.
- the recognition of the functions must require a minimum of processor and memory resources.
- because of the polyvalent architecture of IDA – it supports tens of very different processors – the identification algorithm must be platform-independent, i.e. it must work with the programs compiled for any processor.
- the main() function should be identified and properly labelled as the library' startup-code is of no interest.

The Difficulties

Memory usage

The main obstacle to recognition and identification is the sheer quantity of functions and the size of the memory they occupy. If we evaluate the size of the memory occupied by all _versions_ of all libraries produced by all compiler _vendors_ for memory _models_, we easily fall into the tens of gigabytes range.

Matters get even worse if we try to take OWL, MFC, MFC and similar libraries into account. The storage needed is huge. At this time, personal computers' users can't afford to set aside hundreds of Megabytes of disk space for a simple utility disassembler. Therefore, we had to find an algorithm that diminishes the size of the information needed to recognize standard library functions. Of course, the number of functions that should be recognized dictates the need for an efficient recognition algorithm : a simple brute force search is not an option.

Variability

An additional difficulty arises from the presence of _variant_ bytes in the program. Some bytes are corrected (fixed up) at load time, others become constants at link time, but most of the variant bytes originate from references to external names. In that case the compiler does not know the addresses of the called functions and leaves these bytes equal to zeroes. This so called "fixup information" is usually written to a table in the output file (sometimes called "relocation table" or "relocation information"). The example below

```
B8 0000s                             mov     ax, seg _OVRGROUP_
9A 00000000se                        call    _farmalloc
26: 3B 1E 0000e                      cmp     bx, word ptr es:__ovrbuffer
```

contains variant bytes. The linker will try to resolve external references, replacing zeroes with the addresses of called functions, but some bytes will stay untouched : references to dynamic libraries or bytes containing absolute address in the program. These references can be resolved only at load time by the system loader. It will try to resolve all external references and replace zeroes with absolute addresses. When the system loader cannot resolve an external referencel, as it is the case when the program refers to an unknown DLL, the program will simply not run.
Optimizations introduced by some linkers will also complicate the matter because constant bytes will sometim es be changed. For example:

```
              0000: 9A........        call    far ptr xxx

            is replaced by

              0000: 90                nop
              0001: 0E                push    cs
              0002: E8....            call    near ptr xxx
```

The program will execute as usual, but the replacement introduced by the linker effectively prohibits byte-to-byte comparison with a function template. The presence of variant bytes in a program makes the use of simple checksums for recognition impossible. If functions did not contain variant bytes, the CRC of the first N Bytes would be enough to identify and select a group of functions in a hash table. The use of such tables would greatly decrease the size of the information required for identification : the name of a function, its length and checksum would suffice.

We have already mentionned the fact that the recognition of all standard library functions was not possible or even desirable. One additional proof is the fact that some identical functions do exactly the same thing but are called in a different manner. For example, the functions strcmp() and fstrcmp() are identical in large memory models.

We face a dilemna here : we do not want to discard these functions from the recgnition process since they are not trivial and their labelling would help the user but, we are unable to distinguish them.

And there is more : consider this

```
                call    xxx
                ret
        or
                jmp     xxx
```

At first sight, these pieces of code are not interesting. The problem is that they are present, sometimes in significant number, in standard libraries. The libraries for the Borland C++ v1.5 OS/2 compiler contains 20 calls of this type, in important functions such as read(), write(), etc. Plain comparison of these functions yields nothing. The only way to distinguish those functions is to discover what other function they call. Generally, all short functions (consisting merely of 2-3 instructions) are difficult to recognize and the probability of wrong recognition is very high. However not recognizing them is undesirable, as it can lead to cascade failures : if we do not recognize the function tolower(), we may fail to recognize strlwr() which refers to tolower().

Copyright

Finally, there is an obvious copyright problem : standard libraries may simply not be distributed with a disassembler.

The idea

To address those issues, we created a database of all the functions from all libraries we wanted to recognize. IDA now checks, at each byte of the program being disassembled, whether this byte can mark the start of a standard library function.

The information required by the recognition algorithm is kept in a **signature** file. Each function is represented by a **pattern**. Patterns are first 32 bytes of a function where all variant bytes are marked.

For example:

```
558BEC0EFF7604..........59595DC3558BEC0EFF7604..........59595DC3 _registerbgidriver
558BEC1E078A66048A460E8B5E108B4E0AD1E9D1E980E1C0024E0C8A6E0A8A76 _biosdisk
558BEC1EB41AC55604CD211F5DC3.................................... _setdta
558BEC1EB42FCD210653B41A8B5606CD21B44E8B4E088B5604CD219C5993B41A _findfirst
```

where variant bytes are displayed as ".." Several functions start with the same byte sequence. Therefore a tree structure seems particularly well suited to the storage of those functions :

```
558BEC
      0EFF7604..........59595DC3558BEC0EFF7604..........59595DC3 _registerbgidriver
      1E
        078A66048A460E8B5E108B4E0AD1E9D1E980E1C0024E0C8A6E0A8A76 _biosdisk
        B4
          1AC55604CD211F5DC3                                      _setdta
          2FCD210653B41A8B5606CD21B44E8B4E088B5604CD219C5993B41A   _findfirst
```

Sequences of bytes are kept in the nodes of the tree. In this example, the root of the tree contains the sequence "558BEC", three subtrees stem from the root, respectively starting with bytes 0E, 1E, B4. The subtree starting with B4 gives birth to two subtrees. Each subtree ends with leaves . The information about the function is kept in that (only the name is visible in the above example).

The tree data structure simultaneously achieves two goals :

- Memory requirements are decreased since we store bytes common to several functions in tree nodes. This saving is, of course, proportional to the number of functions starting with the same bytes.
- It is well suited to fast fast pattern matching. The number of comparisons required to match a specific location within a program to all functions in a signature file grows logarithmically with the number of functions.

It would not be very wise to take a decision based on the first 32 bytes of a function alone. As already suggested, modern real-world libraries contain several functions starting with the same bytes:

```
558BEC
      56
        1E
          B8....8ED8
                   33C050FF7608FF7606.........83C406
                                                  8BF083FEFF
                    0. _chmod   (20 5F33)
                    1. _access  (18 9A62)
```

When two functions have the same first 32 bytes, they are stored in the same leaf of the tree. To resolve that situation, we calculate the CRC16 of the bytes starting from position 33 until till the first variant byte. The CRC is stored in the signature file. The number of bytes used to calculate that CRC also needs to be saved, as it differs from function to function. In the above example, the CRC16 is calculated on 20 bytes for the _chmod (bytes 33..52) function and 18 _access function.
There is, of course, a possibility that the first variant byte will be at the 33d position. The length of the sequence of bytes used to calculate the CRC16 is then equal to zero. In practice, this happens rarely and this algorithm gives very low number of false recognitions.

Sometimes functions have the same initial 32-byte pattern and the same CRC16, as in the example below

```
05B8FFFFEB278A4606B4008BD8B8....8EC0
          0. _tolower (03 41CB) (000C:00)
          1. _toupper (03 41CB) (000C:FF)
```

We are unlucky: only 3 bytes were used to calculate the CRC16 and they were the same for both functions. In this case we will try to find a position at which all functions in a leaf have different bytes. (in our example this position is 32+3+000C)
But even this method does not allow to recognize all functions. Here is another example:

```
... (partial tree is shown here)
                0D8A049850E8....83C402880446803C0075EE8BC7:
                  0. _strupr (04 D19F) (REF 0011: _toupper)
                  1. _strlwr (04 D19F) (REF 0011: _tolower)
```

These functions are identical at non-variant bytes and differ only by the functions they call. In this example the only way to distinguish functions is to examine the name referenced from the instruction at offset 11.
The last method has a disadvantage: proper recognition of functions _strupr() and _strlwr() depends on the recognition of functions _toupper() and _tolower(). It means that in the case of failure because of the absence of reference to _toupper() or _tolower() we should defer recognition and repeat it later, after finding _tolower() or _toupper(). This has an impact on the general design of the algorithm : we need a second pass to resolve those deferred recognitions. Luckily, subsequent passes are only applied to a few locations in the program.

Finally, one can find functions that are identical in non-variant bytes, refer to the same names but are called differently. Those functions have the same implementation but different names. Surprisingly, this is a frequent situation in standard libraries, especially in C++ libraries.

We call this situation a _collision_ which occurs when functions attached to a leaf cannot be distinguished from each other by using the described methods. A classical example is:

```
558BEC1EB441C55606CD211F720433C0EB0450E8....5DCB................
   0. _remove (00 0000)
   1. _unlink (00 0000)

   or
8BDC36834702FAE9....8BDC36834702F6E9............................
   0. @iostream@$vsn            (00 0000)
   1. @iostream_withassign@$vsn (00 0000)
```

Artificial Intelligence is the only way to resolve those cases. Since our goal was efficiency and speed, we decided to leave artificial intelligence for the future developments of the algorithm.

The Implementation

In IDA version 3.6, the practical implementation of the algorithm matches the above description almost perfectly. We have limited ourselves to the C and C++ language but it will be, without doubt, possible to write pre-processors for other libraries in the future.

A separate signature file is provided for each compiler. This segregation decreases the probability of cross-compiler identification collisions. A special signature file, called startup signature file is applied to the entry point of the disassembled program to determine the generating compiler. Once it has been identified, we know which signature file should be used for the rest of the disassembly. Our algorithm successfully discerns the startup modules of most popular compilers on the market.

Since we store all functions' signatures for one compiler in one signature file, it is not necessary to discriminate the memory models (small,compact, medium, large, huge) of the libraries and/or versions of the compilers.

We use special startup-signatures for every format of disassembled file. The signature exe.sig is used for programs running under MS DOS, lx.sig or ne.sig – for OS/2, etc.

To decrease a probability of false recognition of short functions, we must absolutely remember any reference to an external name if such a reference exists. It may decrease, to some degree, the probability of the recognition of the function in general but we believe that such an approach is justified. It is better not to recognize than to recognize wrongly. Short functions (shorter than 4 bytes) that do no contain references to external names are not used in the creation of a signature file and no attempt is made to recognize such functions.

The functions from <ctype.h> are short and refer to the array of types of the symbols, therefore we decided to consider the references to this array as an exception : we calculate the CRC16 of the array of the types of the symbols and store it in the signature file.

Without artificial intelligence, the collisions are solved by natural intelligence. The human creator of a signature file chooses the functions to include and to discard from the signature file. This choice is very easy and is practically implemented by the edition of a text file.

The patterns of the functions are not stored in a signature file under their original form (i.e., they do not look like the example figures). In place of the patterns, we store the arrays of bits determining the changing bytes and the values of the individual bytes are stored. Therefore the signature file contains **no byte from the original libraries**, except for the names of the functions. The creation of a signature file involves in 2 stages: the preprocessing of the libraries and the creation of a signature file. In the first stage the program 'parselib' is used. It preprocesses *.obj and *.lib files to produce a pattern-file. The pattern-file contains the patterns of the functions, their names, their CRC16 and all other information necessary to create the signature file. At the second stage the 'sigmake' program builds the signature file from the pattern-file.

This division into 2 stages allows sigmake utility to be independent of the format of the input file. Therefore it will be possible to write other preprocessors for files differing from *.obj and *.lib in future.

We decided to compress (using the InfoZip algorithm) the created signature files to decrease the disk space necessary for their storage.

For the sake of user's convenience we attempted to recognize the main() function as often as it was possible. The algorithm for identifying this function differs from compiler to compiler and from program to program. (DOS/OS2/Windows/GUI/Console...).

This algorithm is written, as a text string, in a signature file. Unfortunately we have not yet been able to automate the creation of this algorithm.

The Results

As it turns out the signature files compress well; they may be compressed by a factor bigger than 2. The reason of this compressibility is that about 95% of a signature file are function names. (Example: the signature file for MFC 2.x was 2.5MB before compression, 700Kb after. It contains 33634 function names; an average of 21 bytes is stored per function. Generally, the ratio of the size of a library size to the size of a signature file varies from 100 to 500.

The percentage of properly recognized functions is very high. Our algorithm recognized all but one function of the "Hello World" program. The unrecognized function consists of only one instruction:

```
        jmp     off_1234
```

We were especially pleased with the fact that there was no false recognition. However it does not mean that they will not occur in the future. It should be noted that the algorithm only works with functions.
Data is sometimes located in the code segment and therefore we need to mark some names as "data names", not as "function names". It is not easy to examine all names in a modern large library and mark all data names.

The implementation of these data names is planned, some time in the future.