12

LIBRARY RECOGNITION USING FLIRT SIGNATURES

At this point it is time to start moving beyond IDA's more obvious capabilities and begin our exploration of what to do after "The initial autoanalysis has been finished."[1] In this chapter we discuss techniques for recognizing standard code sequences such as the library code contained in statically linked binaries or standard initialization and helper functions inserted by compilers.

When you set out to reverse engineer any binary, the last thing that you want to do is waste time reverse engineering library functions whose behavior you could learn much more easily simply by reading a man page, reading some source code, or doing a little Internet research. The challenge presented by statically linked binaries is that they blur the distinction between application code and library code. In a statically linked binary, entire libraries

[1] IDA generates this message in the message window when it has finished its automated processing of a newly loaded binary.

are combined with application code to form a single monolithic executable file. Fortunately for us, tools are available that enable IDA to recognize and mark library code, allowing us to focus our attention on the unique code within the application.

Fast Library Identification and Recognition Technology

Fast Library Identification and Recognition Technology, better known as FLIRT,[2] encompasses the set of techniques employed by IDA to identify sequences of code as library code. At the heart of FLIRT are pattern-matching algorithms that enable IDA to quickly determine whether a disassembled function matches one of the many signatures known to IDA. The *<IDADIR>/sig* directory contains the signature files that ship with IDA. For the most part, these are libraries that ship with common Windows compilers, though a few non-Windows signatures are also included.

Signature files utilize a custom format in which the bulk of the signature data is compressed and wrapped in an IDA-specific header. In most cases, signature filenames fail to give a clear indication of which library the associated signatures were generated from. Depending on how they were created, signature files may contain a library name comment that describes their contents. If we view the first few lines of extracted ASCII content from a signature file, this comment is often revealed. The following Unix-style command[3] generally reveals the comment in the second or third line of output:

```
# strings sigfile | head -n 3
```

Within IDA, there are two ways to view comments associated with signature files. First, you can access the list of signatures that have been applied to a binary via View ▸ Open Subviews ▸ Signatures. Second, the list of all signature files is displayed as part of the manual signature application process, which is initiated via File ▸ Load File ▸ FLIRT Signature File.

Applying FLIRT Signatures

When a binary is first opened, IDA attempts to apply special signature files, designated as startup signatures, to the entry point of the binary. It turns out that the entry point code generated by various compilers is sufficiently different that matching entry point signatures is a useful technique for identifying the compiler that may have been used to generate a given binary.

[2] Please see *http://www.hex-rays.com/idapro/flirt.htm*.

[3] The `strings` command was discussed in Chapter 2, while the `head` command is used to view only the first few lines (three in the example) of its input source.

MAIN VS. _START

Recall that a program's entry point is the address of the first instruction that will be executed. Many longtime C programmers incorrectly believe that this is the address of the function named `main`, when in fact it is not. The file type of the program, *not* the language used to create the program, dictates the manner in which command-line arguments are provided to a program. In order to reconcile any differences between the way the loader presents command-line arguments and the way the program expects to receive them (via parameters to `main`, for example), some initialization code must execute prior to transferring control to `main`. It is this initialization that IDA designates as the entry point of the program and labels `_start`.

This initialization code is also responsible for any initialization tasks that must take place before `main` is allowed to run. In a C++ program, this code is responsible for ensuring that constructors for globally declared objects are called prior to execution of `main`. Similarly, cleanup code is inserted that executes after `main` completes in order to invoke destructors for all global objects prior to the actual termination of the program.

If IDA identifies the compiler used to create a particular binary, then the signature file for the corresponding compiler libraries is loaded and applied to the remainder of the binary. The signatures that ship with IDA tend to be related to proprietary compilers such as Microsoft Visual C++ or Borland Delphi. The reason behind this is that a finite number of binary libraries ship with these compilers. For open source compilers, such as GNU gcc, the binary variations of the associated libraries are as numerous as the operating systems the compilers ship with. For example, each version of FreeBSD ships with a unique version of the C standard library. For optimal pattern matching, signature files would need to be generated for each different version of the library. Consider the difficulty in collecting every variation of *libc.a*[4] that has shipped with every version of every Linux distribution. It simply is not practical. In part, these differences are due to changes in the library source code that result in different compiled code, but huge differences also result from the use of different compilation options, such as optimization settings and the use of different compiler versions to build the library. The net result is that IDA ships with very few signature files for open source compiler libraries. The good news, as you shall soon see, is that Hex-Rays makes tools available that allow you to generate your own signature files from static libraries.

So, under what circumstances might you be required to manually apply signatures to one of your databases? Occasionally IDA properly identifies the compiler used to build the binary but has no signatures for the related compiler libraries. In such cases, either you will need to live without signatures, or you will need to obtain copies of the static libraries used in the binary and generate your own signatures. Other times, IDA may simply fail to identify a compiler, making it impossible to determine which signatures should be

[4] *libc.a* is the version of the C standard library used in statically linked binaries on Unix-style systems.

applied to a database. This is common when analyzing obfuscated code in which the startup routines have been sufficiently mangled to preclude compiler identification. The first thing to do, then, would be to de-obfuscate the binary sufficiently before you could have any hope of matching any library signatures. We will discuss techniques for dealing with obfuscated code in Chapter 21.

Regardless of the reason, if you wish to manually apply signatures to a database, you do so via File ▸ Load File ▸ FLIRT Signature File, which opens the signature selection dialog shown in Figure 12-1.

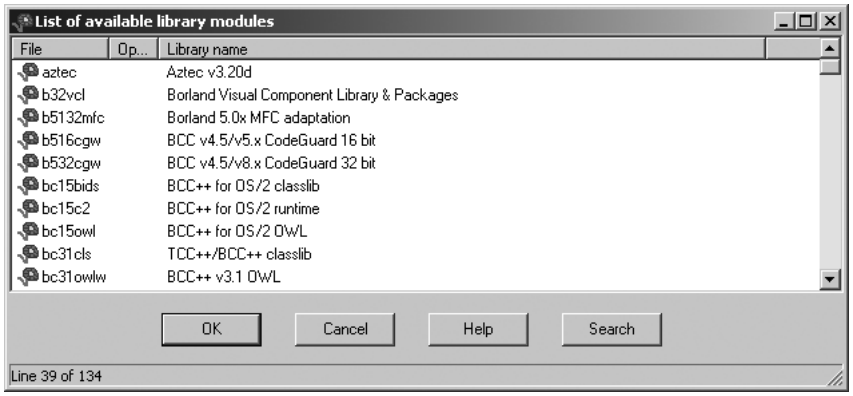

Figure 12-1: FLIRT signature selection

The File column reflects the name of each *.sig* file in IDA's *<IDADIR>/sig* directory. Note that there is no means to specify an alternate location for *.sig* files. If you ever generate your own signatures, they need to be placed into *<IDADIR>/sig* along with every other *.sig* file. The Library name column displays the library name comment that is embedded within each file. Keep in mind that these comments are only as descriptive as the creator of the signatures (which could be you!) chooses to make them.

When a library module is selected, the signatures contained in the corresponding *.sig* file are loaded and compared against every function within the database. Only one set of signatures may be applied at a time, so you will need to repeat the process if you wish to apply several different signature files to a database. When a function is found to match a signature, the function is marked as a library function, and the function is automatically renamed according to the signature that has been matched.

WARNING *Only functions named with an IDA dummy name can be automatically renamed. In other words, if you have renamed a function, and that function is later matched by a signature, then the function will not be renamed as a result of the match. Therefore, it is to your benefit to apply signatures as early in your analysis process as possible.*

Recall that statically linked binaries blur the distinction between application code and library code. If you are fortunate enough to have a statically linked binary that has not had its symbols stripped, you will at least have useful function names (as useful as the trustworthy programmer has chosen

to create) to help you sort your way through the code. However, if the binary has been stripped, you will have perhaps hundreds of functions, all with IDA-generated names that fail to indicate what the function does. In both cases, IDA will be able to identify library functions only if signatures are available (function names in an unstripped binary do not provide IDA with enough information to definitively identify a function as a library function). Figure 12-2 shows the Overview Navigator for a statically linked binary.

Figure 12-2: Statically linked with no signatures

In this display, no functions have been identified as library functions, so you may find yourself analyzing far more code than you really need to. After application of an appropriate set of signatures, the Overview Navigator is transformed as shown in Figure 12-3.

Figure 12-3: Statically linked binary with signatures applied

As you can see, the Overview Navigator provides the best indication of the effectiveness of a particular set of signatures. With a large percentage of matched signatures, substantial portions of code will be marked as library code and renamed accordingly. In the example in Figure 12-3, it is highly likely that the actual application-specific code is concentrated in the far-left portion of the navigator display.

There are two points worth remembering when applying signatures. First, signatures are useful even when working with a binary that has not been stripped, in which case you are using signatures more to help IDA identify library functions than to rename those functions. Second, statically linked binaries may be composed of several separate libraries, requiring the application of several sets of signatures in order to completely identify all library functions. With each additional signature application, additional portions of the Overview Navigator will be transformed to reflect the discovery of library code. Figure 12-4 shows one such example. In this figure, you see a binary that was statically linked with both the C standard library and the OpenSSL[5] cryptographic library.

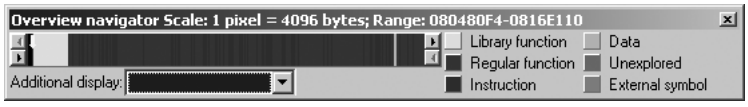

Figure 12-4: Static binary with first of several signatures applied

[5] Please see *http://openssl.org/*.

Specifically, you see that following application of the appropriate signatures for the version of OpenSSL in use in this application, IDA has marked a small band (the lighter band toward the left edge of the address range) as library code. Statically linked binaries are often created by taking the application code first and then appending required libraries to create the resulting executable. Given this picture, we can conclude that the memory space to the right of the OpenSSL library is likely occupied by additional library code, while the application code is most likely in the very narrow band to the left of the OpenSSL library. If we continue to apply signatures to the binary shown in Figure 12-4, we eventually arrive at the display of Figure 12-5.

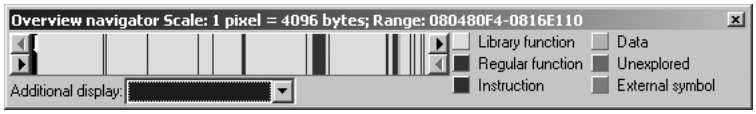

Figure 12-5: Static binary following application of several signatures

In this example, we have applied signatures for *libc*, *libcrypto*, *libkrb5*, *libresolv*, and others. In some cases we selected signatures based on strings located within the binary; in other cases we chose signatures based on their close relationship to other libraries already located within the binary. The resulting display continues to show a dark band in the right half of the navigation band and a smaller dark band at the extreme left edge of the navigation band. Further analysis is required to determine the nature of these remaining nonlibrary portions of the binary. In this case we would learn that the wider dark band on the right side is part of an unidentified library, while the dark band on the left is the application code.

Creating FLIRT Signature Files

As we discussed previously, it is simply impractical for IDA to ship with signature files for every static library in existence. In order to provide IDA users with the tools and information necessary to create their own signatures, Hex-Rays distributes the Fast Library Acquisition for Identification and Recognition (FLAIR) tool set. The FLAIR tools are made available on your IDA distribution CD or via download from the Hex-Rays website[6] for authorized customers. Like several other IDA add-ons, the FLAIR tools are distributed in a Zip file. For IDA version 5.2, the associated FLAIR tools are contained in *flair52.zip*. Hex-Rays does not necessarily release a new version of the FLAIR tools with each version of IDA, so you should use the most recent version of FLAIR that does not exceed your version of IDA.

Installation of the FLAIR utilities is a simple matter of extracting the contents of the associated Zip file, though we highly recommend that you create a dedicated *flair* directory as the destination because the Zip file is not

[6] The current version is *flair52.zip* and is available here: *http://www.hex-rays.com/idapro/ida/flair52.zip*. A username and password supplied by Hex-Rays are required to access the download.

organized with a top-level directory. Inside the FLAIR distribution you will find several text files that constitute the documentation for the FLAIR tools. Files of particular interest include these:

readme.txt
This is a top-level overview of the signature-creation process.

plb.txt
This file describes the use of the static library parser, *plb.exe*. Library parsers are discussed in more detail in "Creating Pattern Files" on page 219.

pat.txt
This file details the format of pattern files, which represent the first step in the signature-creation process. Pattern files are also described in "Creating Pattern Files" on page 219.

sigmake.txt
This file describes the use of *sigmake.exe* for generating *.sig* files from pattern files. Please refer to "Creating Signature Files" on page 221 for more details.

Additional top-level content of interest includes the *bin* directory, which contains all of the FLAIR tools executable files, and the *startup* directory, which contains pattern files for common startup sequences associated with various compilers and their associated output file types (PE, ELF, and so on). An important point to understand regarding the FLAIR tools is that while all of the tools run only from the Windows command prompt, the resulting signature files may be used with all IDA variants (Windows, Linux, and OS X).

Signature-Creation Overview

The basic process for creating signatures files does not sound complicated, as it boils down to four simple-sounding steps.

1. Obtain a copy of the static library for which you wish to create a signature file.
2. Utilize one of the FLAIR parsers to create a pattern file for the library.
3. Run *sigmake.exe* to process the resulting pattern file and generate a signature file.
4. Install the new signature file in IDA by copying it to *<IDADIR>/sig*.

Unfortunately, in practice, only the last step is as easy as it sounds. In the following sections, we discuss the first three steps in more detail.

Identifying and Acquiring Static Libraries

The first step in the signature-generation process is to locate a copy of the static library for which you wish to generate signatures. This can pose a bit of a challenge for a variety of reasons. The first obstacle is to determine which library you actually need. If the binary you are analyzing has not been stripped,

you might be lucky enough to have actual function names available in your disassembly, in which case Google will probably provide several pointers to likely candidates.

Stripped binaries are not quite as forthcoming regarding their origins. Lacking function names, you may find that a good `strings` search may yield sufficiently unique strings to allow for library identification, such as the following, which is a dead giveaway:

```
OpenSSL 0.9.8a 11 Oct 2005
```

Copyright notices and error strings are often sufficiently unique that once again you can use Google to narrow your search. If you choose to run `strings` from the command line, remember to use the `-a` option to force `strings` to scan the entire binary; otherwise you may miss some potentially useful string data.

In the case of open source libraries, you are likely to find source code readily available. Unfortunately, while the source code may be useful in helping you understand the behavior of the binary, you cannot use it to generate your signatures. It might be possible to use the source to build your own version of the static library and then use that version in the signature-generation process. However, in all likelihood, variations in the build process will result in enough differences between the resulting library and the library you are analyzing that any signatures you generate will not be terribly accurate.

The best option is to attempt to determine the exact origin of the binary in question. By this we mean the exact operating system, operating system version, and distribution (if applicable). Given this information, the best option for creating signatures is to copy the libraries in question from an identically configured system. Naturally, this leads to the next challenge: Given an arbitrary binary, on what system was it created? A good first step is to use the `file` utility to obtain some preliminary information about the binary in question. In Chapter 2 we saw some sample output from `file`. In several cases, this output was sufficient to provide likely candidate systems. The following is just one example of very specific output from `file`:

```
$ file sample_file_1
sample_file_1: ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD),
for FreeBSD 5.4, statically linked, FreeBSD-style, stripped
```

In this case we might head straight to a FreeBSD 5.4 system and track down *libc.a* for starters. The following example is somewhat more ambiguous, however:

```
$ file sample_file_2
sample_file_2: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
for GNU/Linux 2.6.9, statically linked, stripped
```

We appear to have narrowed the source of the file to a Linux system, which, given the abundance of available Linux distributions, is not saying much. Turning to `strings` we find the following:

```
GCC: (GNU) 4.1.1 20060525 (Red Hat 4.1.1-1)
```

Here the search has been narrowed to Red Hat distributions (or derivatives) that shipped with gcc version 4.1.1. GCC tags such as this are not uncommon in binaries compiled using gcc, and fortunately for us, they survive the stripping process and remain visible to `strings`.

Keep in mind that the `file` utility is not the be all and end all in file identification. The following output demonstrates a simple case in which `file` seems to know the type of the file being examined but for which the output is rather nonspecific.

```
$ file sample_file_3
sample_file_3: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), stripped
```

This example was taken from a Solaris 10 x86 system. Here again, the `strings` utility might be useful in pinpointing this fact.

Creating Pattern Files

At this point you should have one or more libraries for which you wish to create signatures. The next step is to create a pattern file for each library. Pattern files are created using an appropriate FLAIR parser utility. Like executable files, library files are built to various file format specifications. FLAIR provides parsers for several popular library file formats. As detailed in FLAIR's *readme.txt* file, the following parsers can be found in FLAIR's *bin* directory:

plb.exe
Parser for OMF libraries (commonly used by Borland compilers)

pcf.exe
Parser for COFF libraries (commonly used by Microsoft compilers)

pelf.exe
Parser for ELF libraries (found on many Unix systems)

ppsx.exe
Parser for Sony PlayStation PSX libraries

ptmobj.exe
Parser for TriMedia libraries

pomf166.exe
Parser for Kiel OMF 166 object files

To create a pattern file for a given library, specify the parser that corresponds to the library's format, the name of the library you wish to parse, and the name of the resulting pattern file that should be generated. For a copy of *libc.a* from a FreeBSD 6.1 system, you might use the following:

```
$ ./pelf libc.a libc_FreeBSD61.pat
libc.a: skipped 0, total 986
```

Here, the parser reports the file that was parsed (*libc.a*), the number of functions that were skipped (0),[7] and the number of signature patterns that were generated (986). Each parser accepts a slightly different set of command-line options documented only through the parser's usage statement. Executing a parser with no arguments displays the list of command-line options accepted by that parser. The *plb.txt* file contains more detailed information on the options accepted by the plb.*exe* parser. This file is a good basic source of information, since other parsers accept many of the options it describes as well. In many cases, simply naming the library to be parsed and the pattern file to be generated is sufficient.

A pattern file is a text file that contains, one per line, the extracted patterns that represent functions within a parsed library. A few lines from the pattern file created previously are shown here:

```
5589E58B55108B450C8B4D0885D2EB06890183C1044A75F88B4508C9C3..... 00 0000 001D :0000 _wmemset
5589E58B4D1057C1E102568B7D088B750CFCC1E902F3A55E8B45085FC9C3.... 00 0000 001E :0000 _wmemcpy
5589E556538B751031DB39F38B4D088B550C73118B023901751183C10483C204 19 A9BE 0039 :0000 _wmemcmp
```

The format of an individual pattern is described in FLAIR's *pat.txt* file. In a nutshell, the first portion of a pattern lists the initial byte sequence of the function to a maximum of 32 bytes. Allowance is made for bytes that may vary as a result of relocation entries. Such bytes are displayed using two dots. Dots are also used to fill the pattern out to 64[8] characters when a function is shorter than 32 bytes (as `_wmemset` is in the previous code). Beyond the initial 32 bytes, additional information is recorded to provide more precision in the signature-matching process. Additional information encoded into each pattern line includes a CRC16[9] value computed over a portion of the function, the length of the function in bytes, and a list of symbol names referenced by the function. In general, the longer functions that reference many other symbols yield more complex pattern lines. In the file *libc_FreeBSD61.pat* generated previously, some pattern lines exceed 20,000 characters in length.

[7] The plb and pcf parsers may skip some functions depending on the command-line options supplied to the parsers and the structure of the library being parsed.

[8] At two characters per byte, 64 hexadecimal characters are required to display the contents of 32 bytes.

[9] This is a 16-bit cyclic redundancy check value. The CRC16 implementation utilized for pattern generation is included with the FLAIR tool distribution in the file *crc16.cpp*.

Several third-party programmers have created utilities designed to generate patterns from existing IDA databases. One such utility is IDB_2_PAT,[10] an IDA plug-in written by J.C. Roberts that is capable of generating patterns for one or more functions in an existing database. Utilities such as these are useful if you expect to encounter similar code in additional databases and have no access to the original library files used to create the binary being analyzed.

Creating Signature Files

Once you have created a pattern file for a given library, the next step in the signature-creation process is to generate a *.sig* file suitable for use with IDA. The format of an IDA signature file is substantially different from a pattern file. Signature files utilize a proprietary binary format designed both to minimize the amount of space required to represent all of the information present in a pattern file and to allow for efficient matching of signatures against actual database content. A high-level description of the structure of a signature file is available on the Hex-Rays website.[11]

FLAIR's *sigmake.exe* utility is used to create signature files from pattern files. By splitting pattern generation and signature generation into two distinct phases, the signature-generation process is completely independent of the pattern-generation process, which allows for the use of third-party pattern generators. In its simplest form, signature generation takes place by using *sigmake.exe* to parse a *.pat* file and create a *.sig* file, as shown here:

```
$ ./sigmake libssl.pat libssl.sig
```

If all goes well, a *.sig* file is generated and ready to install into *<IDADIR>/sig*. However, the process seldom runs that smoothly.

NOTE *The* `sigmake` *documentation file,* sigmake.txt*, recommends that signature filenames follow the MS-DOS 8.3 name-length convention. This is not a hard-and-fast requirement, however. When longer filenames are used, only the first eight characters of the base filename are displayed in the signature-selection dialog.*

Signature generation is often an iterative process, as it is during this phase when *collisions* must be handled. A collision occurs any time two functions have identical patterns. If collisions are not resolved in some manner, it is not possible to determine which function is actually being matched during the signature-application process. Therefore, `sigmake` must be able to resolve each generated signature to exactly one function name. When this is not possible, based on the presence of identical patterns for one or more functions, `sigmake` refuses to generate a *.sig* file and instead generates an *exclusions*

[10] Please see *http://www.openrce.org/downloads/details/26/IDB_2_PAT.*

[11] Please see *http://www.hex-rays.com/idapro/flirt.htm.*

file (*.exc*). A more typical first pass using `sigmake` and a new *.pat* file (or set of *.pat* files) might yield the following.

```
$ ./sigmake libc_FreeBSD61.pat libc_FreeBSD61.sig
See the documentation to learn how to resolve collisions.
: modules/leaves: 13443631/970, COLLISIONS: 911
```

The documentation being referred to is *sigmake.txt,* which describes the use of `sigmake` and the collision-resolution process. In reality, each time `sigmake` is executed, it searches for a corresponding exclusions file that might contain information on how to resolve any collisions that `sigmake` may encounter while processing the named pattern file. In the absence of such an exclusions file, and when collisions occur, `sigmake` generates such an exclusions file rather than a signature file. In the previous example, we would find a newly created file named *libc_FreeBSD61.exc*. When first created, exclusions files are text files that detail the conflicts that `sigmake` encountered while processing the pattern file. The exclusions file must be edited to provide `sigmake` with guidance as to how it should resolve any conflicting patterns. The general process for editing an exclusions file follows.

When generated by `sigmake`, all exclusions files begin with the following lines:

```
;--------- (delete these lines to allow sigmake to read this file)
; add '+' at the start of a line to select a module
; add '-' if you are not sure about the selection
; do nothing if you want to exclude all modules
```

The intent of these lines it to remind you what to do to resolve collisions before you can successfully generate signatures. The most important thing to do is delete the four lines that begin with semicolons, or `sigmake` will fail to parse the exclusions file during subsequent execution. The next step is to inform `sigmake` of your desire for collision resolution. A few lines extracted from *libc_FreeBSD61.exc* appear here:

```
___ntohs            00 0000 0FB744240486C4C3.........................................
___htons            00 0000 0FB744240486C4C3.........................................

_index              00 0000 538B4424088A4C240C908A1838D974074084DB75F531C05BC3.............
_strchr             00 0000 538B4424088A4C240C908A1838D974074084DB75F531C05BC3.............

_rindex             00 0000 538B5424088A4C240C31C0908A1A38D9750289D04284DB75F35BC3.........
_strrchr            00 0000 538B5424088A4C240C31C0908A1A38D9750289D04284DB75F35BC3.........
```

These lines detail three separate collisions. In this case, we are being told that the function `ntohs` is indistinguishable from `htons`, `index` has the same signature as `strchr`, and `rindex` collides with `strrchr`. If you are familiar with any of these functions, this result may not surprise you, as the colliding functions are essentially identical (for example, `index` and `strchr` perform the same action).

In order to leave you in control of your own destiny, sigmake expects you to designate no more than one function in each group as the proper function for the associated signature. You select a function by prefixing the name with a plus character (+) if you want the name applied anytime the corresponding signature is matched in a database or a minus character (-) if you simply want a comment added to the database whenever the corresponding signature is matched. If you do not want any names applied when the corresponding signature is matched in a database, then you do not add any characters. The following listing represents one possible way to provide a valid resolution for the three collisions noted previously:

```
+___ntohs           00 0000 0FB744240486C4C3........................................
___htons            00 0000 0FB744240486C4C3........................................

_index              00 0000 538B4424088A4C240C908A1838D974074084DB75F531C05BC3......
_strchr             00 0000 538B4424088A4C240C908A1838D974074084DB75F531C05BC3......

_rindex             00 0000 538B5424088A4C240C31C0908A1A38D9750289D04284DB75F35BC3..
-_strrchr           00 0000 538B5424088A4C240C31C0908A1A38D9750289D04284DB75F35BC3..
```

In this case we elect to use the name ntohs whenever the first signature is matched, do nothing at all when the second signature is matched, and have a comment about strrchr added when the third signature is matched. The following points are useful when attempting to resolve collisions:

1. To perform minimal collision resolution, simply delete the four commented lines at the beginning of the exclusions file.
2. Never add a +/- to more than one function in a collision group.
3. If a collision group contains only a single function, *do not* add a +/- in front of that function; simply leave it alone.
4. Subsequent failures of sigmake cause data, including comment lines, to be appended to any existing exclusions file. This extra data should be removed and the original data corrected (if the data was correct, sigmake would not have failed a second time) before rerunning sigmake.

Once you have made appropriate changes to your exclusions file, you must save the file and rerun sigmake using the same command-line arguments that you used initially. The second time through, sigmake should locate, and abide by, your exclusions file, resulting in the successful generation of a *.sig* file. Successful operation of sigmake is noted by the lack of error messages and the presence of a *.sig* file, as shown here:

```
$ ./sigmake libc_FreeBSD61.pat libc_FreeBSD61.sig
```

After a signature file has been successfully generated, you make it available to IDA by copying it to your *<IDADIR>/sig* directory. Then your new signatures are available using File ▸ Load File ▸ FLIRT Signature File.

Note that we have purposefully glossed over all of the options that can be supplied to both the pattern generators and `sigmake`. A rundown of available options is provided in *plb.txt* and *sigmake.txt*. The only option we will make note of is the `-n` option used with `sigmake`. This option allows you to embed a descriptive name inside a generated signature file. This name is displayed during the signature-selection process (see Figure 12-1), and it can be very helpful when sorting through the list of available signatures. The following command line embeds the name string "FreeBSD 6.1 C standard library" within the generated signature file:

```
$ ./sigmake -n"FreeBSD 6.1 C standard library" libc_FreeBSD61.pat libc_FreeBSD61.sig
```

As an alternative, library names can be specified using directives within exclusion files. However, since exclusion files may not be required in all signature-generation cases, the command-line option is generally more useful. For further details, please refer to *sigmake.txt*.

Startup Signatures

IDA also recognizes a specialized form of signatures, called *startup signatures*. Startup signatures are applied when a binary is first loaded into a database in an attempt to identify the compiler that was used to create the binary. If IDA can identify the compiler used to build a binary, then additional signature files, associated with the identified compiler, are automatically loaded during the initial analysis of the binary.

Given that the compiler type is initially unknown when a file is first loaded, startup signatures are grouped by and selected according to the file type of the binary being loaded. For example, if a Windows PE binary is being loaded, then startup signatures specific to PE binaries are loaded in an effort to determine the compiler used to build the PE binary in question.

In order to generate startup signatures, `sigmake` processes patterns that describe the startup routine[12] generated by various compilers and groups the resulting signatures into a single type-specific signature file. The `startup` directory in the FLAIR distribution contains the startup patterns used by IDA, along with the script, *startup.bat*, used to create the corresponding startup signatures from those patterns. Refer to *startup.bat* for examples of using `sigmake` to create startup signatures for a specific file format.

In the case of PE files, you would notice several *pe_*.pat* files in the startup directory that describe startup patterns used by several popular Windows compilers, including *pe_vc.pat* for Visual Studio patterns and *pe_gcc.pat* for Cygwin/gcc patterns. If you wish to add additional startup patterns for PE files, you would need to add them to one of the existing PE pattern files or create a new pattern file with a `pe_` prefix in order for the startup signature-generation script to properly find your patterns and incorporate them into the newly generated PE signatures.

[12] The startup routine is generally designated as the program's entry point. In a C/C++ program, the purpose of the startup routine is to initialize the program's environment prior to passing control to the `main` function.

One last note about startup patterns concerns their format, which unfortunately is slightly different from patterns generated for library functions. The difference lies in the fact that a startup pattern line is capable of relating the pattern to additional sets of signatures that should also be applied if a match against the pattern is made. Other than the example startup patterns included in the *startup* directory, the format of a startup pattern is not documented in any of the text files included with FLAIR.

Summary

Automated library code identification is an essential capability that significantly reduces the amount of time required to analyze statically linked binaries. With its FLIRT and FLAIR capabilities, IDA makes such automated code recognition not only possible but extensible by allowing users to create their own library signatures from existing static libraries. Familiarity with the signature-generation process is an essential skill for anyone who expects to encounter statically linked binaries.