Signatures for Library Functions in Executable Files

Mike Van Emmerik
Information Security Research Centre
Queensland University of Technology

Abstract

A method for efficiently generating signatures for detecting library functions in executable files is described. The signatures are used to automatically detect such functions in *dcc*, the reverse compiler at the Queensland University of Technology. Difficulties arise from the variability of the signatures, the multiplicity of library code vendors, and of memory models, and indistinguishable functions. An efficient hashing technique involving perfect optimal hashing functions is used. Performance is good - the signature files are created in a few seconds, and the name of a library function can be found in about the time of two standard hashes. One signature file is required for each vendor, version, and memory model combination, and they are generated from the appropriate library file (e.g. slibce.lib). Some issues are yet to be addressed, such as variation due to floating point math options (e.g. emulator, fast alternate, or coprocessor calls).

1 Application

Signatures are required wherever there is a need to recognise a binary pattern, such as, when detecting viruses. However the signature techniques described in this paper were motivated by the need to identify library signatures in *dcc*, the reverse C compiler currently under development at the Queensland University of Technology. Whenever *dcc* comes across a function call, it can now easily determine if it is a call to a library function, or to a function in the original program. If it is a library function, there is no need to reverse compile it, as attempting to do so would waste time and clutter the output file (the regenerated C source). Furthermore, many library functions are written in assembler and contain instructions and structures that are difficult or impossible to decompile. In this way, small programs decompile to much smaller sources when the irrelevant library code is eliminated. All programs are much more readable when calls to library functions use their symbolic name (e.g. "strcmp" rather than "proc003").

2 Library files

A library file is a file supplied by a compiler vendor to be statically linked with the user's program to implement standard library functions like "writeln" and "printf". Internally, a library file is organised in the same way as a relocatable object file, which is the output of the compiler and the input to the linker. Compiler vendors provide several library files for the various memory models that the compiler supports. There are 6 different memory models for typical MS-DOS compilers; tiny, small, medium, compact, large, and huge. These different models arise from the choice of 16 or 32 bit pointers for code and data; huge model is a special case of large model that can cope with arrays larger than 64k. Other architectures may have such memory models, for example on the 68000 microprocessor it is possible to choose between 16 and 32 bit offsets to data. Thus, a compiler vendor might supply a library file with a name like slibce.lib, being for small model (the 's'), and floating point emulator (the 'e').

The particular library function image that will be found in an executable program will depend on the memory model. This variation is overcome by storing a signature file for each memory model, and selecting the appropriate signature file at run time. Because the different vendors have different libraries, and different versions of the same vendor's compiler may use different libraries, there is a signature file for each combination of vendor, version, and memory model. A naming convention is used, and *dcc* looks for the appropriately named

signature file automatically, based on signatures near the beginning of the file (see section 7 for details). The user need generate only those signature files that he is likely to need.

Some complers do not use standard library .lib files for library code. In these cases, custom programs have to be written to produce signatures. Such a program has been written to read a Turbo Pascal® turbo.tpu file. This file is the equivalent of the .lib file for other languages. Turbo Pascal® programs (after version 4.0) use the large memory model exclusively, so no memory model variations need be considered. However, the library routines differ from one version to the next, so version numbers are considered. Pascal is different from most other languages in that there is not a one to one mapping between action keywords like writeln and library functions that implement them. Consider displaying a string and a number in Pascal and C:

```
writeln('Result: ', i);          printf("Result: %d\n", i);
```

which will ideally generate the following output from *dcc*:

```
writestring("Result: ");         printf("Result: %d\n", loc1);
writeint(loc1);
crlf();
```

3 Wildcards

Even given a particular vendor, version and memory model, the binary pattern of any piece of library code may well depend on where it is loaded, where other modules that it calls are located, and where its data happens to be allocated in the data group. However, some of the bit patterns are constant (e.g. opcodes, opcode prefixes, short jump offsets, indexes to the local stack frame, byte immediate values, etc). So the key to recognising a function is to disassemble it, to determine which parts are invariant. The assembler code of the pattern is not actually generated, but most of the functionality of a disassembler is required to "keep in step" and analyse the addressing modes of the instruction to determine which bytes of the instruction are invariant. For example, in the instruction

```
8A 46 42   MOV  AX, [BP+42]
```

the opcode (8A), the mod/rm byte (46) and the offset (42) are all invariant. However, in the instruction

```
B8 42 42   MOV  AX, 4242
```

only the opcode (B8) is invariant. The immediate operand (4242) may be an immediate constant, or the offset of an item in the data segment. Therefore it may or may not be invariant, and has to be assumed to be variable, since exact analysis by disassembly is never decisive. In the signatures, variable bytes are replaced with a special value called the wildcard; in our case, 0xF4, the opcode for HALT, which would rarely be used as an opcode. So in a signature, the above instruction would be stored as

```
B8 F4 F4.
```

Signatures consist largely of the first n bytes of the function, with variable bytes replaced by the wildcard. The value for n is important - if too small, some functions will not be able to be distinguished. If n is chosen too large, the signatures will take up more space, but more importantly, there will be an increased risk of running into the next library function, or any other code or data linked into the file. The value for n in *dcc* was arbitrarily chosen to be 23; the first three bytes of most functions is a prologue to set up the stack frame, so there are 20 bytes available (less wildcards) for distinguishing functions. Experimental results have shown that this is a reasonable choice for n.

Note that there needs to be protection against running into the next function (see section 4, Chopping the Signatures), as the next piece of code or data may not be guaranteed to follow the library function in question. For example: suppose the library function for "cos" was 10 bytes of code and a jump to the function "sin", but it happened that in the library file the function "strcpy" followed "cos". If the executable program happens not to use "strcpy", then it will not be linked in, and some other library function will be found there. If the signature

for "cos" was 10 bytes followed by the first 13 bytes of "strcpy", then obviously the 23 byte pattern for "cos" would not match the executable file, and the "cos" function would not be recognised.

Before a comparison between the given function and a signature is made, a copy is made of the first n bytes of the function, and the resultant binary fragment is "fixed" for wildcards using the same algorithm used to store the signature. (The signature may also be chopped, as described below). This removes all variability from the pattern, since they are now replaced with the fixed wildcard byte. Therefore, a binary comparison of the signature in the hash table and the fixed copy of the candidate function determines whether a match has been found, i.e. the function is recognised or not. Because a perfect hashing function is used, there is no need to consider other signatures corresponding to the hash of the candidate function.

4 Chopping the signatures

Regardless of the length of signature chosen, some of the functions that are being detected will be shorter. Since it cannot be guaranteed that whatever follows the function will always be the same everywhere the function is found, it is important to find the end of the function, and fill the remainder with a known value (we use zeroes). Thus, random contents will not disturb the signature comparison. Unfortunately, finding the end of a function is equivalent to the halting problem, which is not computable in general. However, it was found that a simple heuristic was sufficient for the purpose. The heuristic is to regard any unconditional branch, or any return from subroutine instruction, as the end of the function. Some signatures will be unnecessarily shortened by this technique, but most of these will be unique even in their shortened form.

Figure 1 depicts a typical signature. Of course, only the binary image (second column) is actually stored; the addresses and disassembly are merely for readability of the example.

```
0100 55            PUSH    BP
0101 8BEC          MOV     BP,SP
0103 56            PUSH    SI
0104 8B7604        MOV     SI,[BP+04]
0107 56            PUSH    SI
0108 E8F4F4        CALL    XXXX       ;Destination "wildcarded"
010B 59            POP     CX
010C 0BC0          OR      AX,AX
010E 7405          JZ      0115
0110 B8F4F4        MOV     AX,XXXX    ;Operand "wildcarded"
0113 EB4C          JMP     0161
0115 0000          ; Padding due to "chopping"
```

Figure 1. Signature for library function "fseek".

5 Hashing the signatures

Given the library file, we need to extract and store these signatures in a table, and need some way of identifying quickly whether a candidate function is in the table or not. Hashing is an obvious technique, and since the keys are constant (the fragments of functions suitably wildcarded and chopped), perfect hashing functions can be used to remove the need for dealing with collisions. Optimal hashing functions would also reduce space requirements, so a perfect optimal hashing algorithm was used [1, 2, 3, 4]. This technique involves random functions, and generates the tables required for the hashing function in random linear time. The hash function itself executes in linear time, being little more than two standard hash functions. The detailed working of the algorithm will not be given here, although examples will indicate the outline of the steps involved.

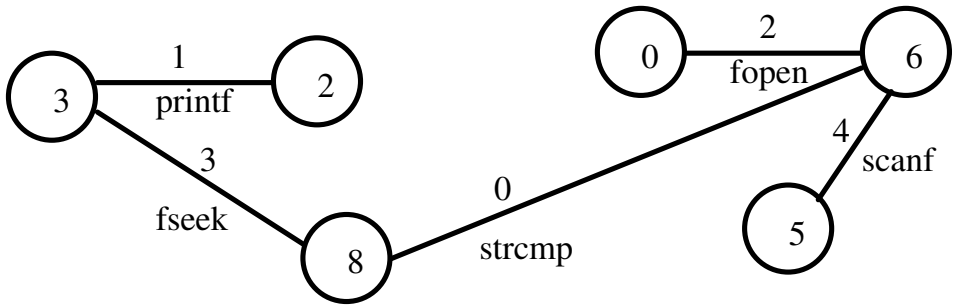

Figure 2. A small example signature graph.

In Figure 2, a sample signature graph is given. The node numbers have no significance, and are selected by random tables. The edge numbers are the indices of the functions, so for example function 0 is "strcmp". Hashes of the input signature choose a pair of vertices, which yields the required index. The graph must be acyclic for this mapping to be unique.

The signature files consist of a prologue and an array of records. The prologue contains the information necessary to generate the tables required by the hash function. The array of records contains the signatures and the symbolic names. For convenience, a fixed length null terminated string is used for the latter.

6 Duplicate keys

It is a fundamental assumption of perfect hashing functions that the input keys are unique. After all, you cannot expect a hash function to return a unique value when the keys themselves are not unique. However, not all library signatures will be unique, since some library functions are not distinct for PCs, e.g. unlink() and remove(). Also, it is likely that chopping or choice of signature size shorter than that required to distinguish all several hundred functions will cause several of the signatures to be duplicates. We shall see that this requires a modification of the function to generate the hashing function tables.

The mapping phase of the algorithm consists of generating random tables, and constructing a graph associated with these tables. The vertices of the graph are chosen by performing an operation on the keys using these tables. (This operation corresponds with half of the ultimate hash function). Edges correspond to the index of the key, i.e. the value required to be supplied by the hashing function. If the graph is cyclic, the values are not suitable, and new random tables are generated. The mapping process is repeated until the graph is acyclic. Regardless of the values in the table, two keys with the same value will always choose the same pair of vertices as each other, and the edges will be distinct (since the key indexes for different keys will always be distinct). Thus, the presence of duplicate keys guarantees unit cycles in the graph, and the mapping algorithm will never terminate. However, we have to live with the fact that not all keys will be unique, so we have to either eliminate some input keys (this will involve some movement in the table), or modify the algorithm to tolerate duplicate keys, but still detect genuine cycles in the graph. The latter technique was chosen, due to its speed and simplicity, and the fact that all signatures are present (even though some will never be recognised, since their duplicates will be recognised in their place). This means that the function recognition process will not be perfect, but this is an acceptable situation, since the intention of library function recognition is merely to aid the end user in understanding the program.

The modification to the algorithm is conceptually simple and fits in nicely with the already present depth first search of the graph. Each candidate edge is tested for a unit cycle (i.e. two or more edges joining distinct vertices). Self loops (one edge terminating in the same vertex) are not caused by duplicate keys, and are tested for as each input key is read, before the graph is traversed. Self loops, like all genuine cycles not involving duplicate keys, cause the current random tables to be abandoned, and to restart the mapping phase. If a candidate edge terminates in the parent vertex, then the keys associated with the candidate edge and the parent edge are compared. (Remember that the edge values are actually key indices, so comparing keys is relatively

simple). If these are the same, the candidate edge is ignored for the purposes of cycle detection. Since the parent edge is still considered for cycle detection, nothing is lost by disregarding this candidate edge.

Such manipulations are facilitated by using an edge oriented representation of the graph [5]. This representation has additional benefits, viz a simple, compact array representation. No dynamically allocated entities need be used, apart from three arrays whose sizes are known in advance.

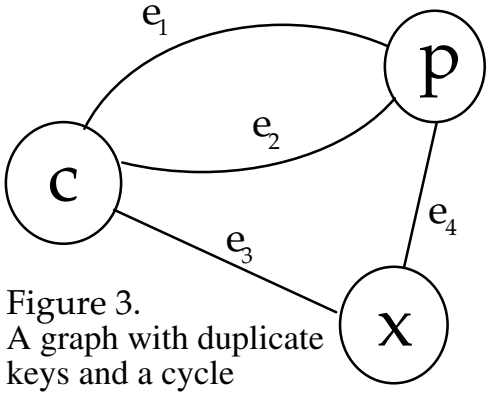

Figure 3.
A graph with duplicate keys and a cycle

In the example graph in Figure 3, the depth first traversal visits vertex c first. Edge e_1 is followed first, leading to vertex p. There are 3 edges associated with vertex p; they are traversed in the order e_1, e_2, e_4. When e_1 is traversed, it is noted that e_1 is the parent edge (a parent edge as well as parent vertex is maintained). Therefore, e_1 is ignored. When following e_2, it is not the parent edge, but leads to the parent vertex. Thus, there is a unit cycle and the keys associated with edges e_1 and e_2 are compared. If different, then this unit cycle proves the graph to be cyclic. Otherwise, e_2 can safely be ignored for the purpose of cycle detection. However, when e_4 and e_3 are processed, the true cycle is found.

7 Choosing the signature file

Given a set of signature files, it is necessary to decide which one to use. This is again a pattern recognition exercise. The vendor is readily identified from the first few bytes of the startup code. *dcc* uses this code to determine the default value for the data segment register. Most compilers set this value in the first few tens of bytes of startup code. Also required by *dcc* is the location of the main program (function main() in a C program, or the main procedure of a Pascal program), since the scanning of functions starts there.

The address of the main program and the memory model are found in *dcc* by testing against several constant signatures stored internally. Wildcards are used, since the offsets in the startup code vary the same as with library code. Only a few hundred bytes of the executable file need be searched for these signatures. Similarly, the compiler vendor is decided by testing against a few static signatures, also stored internally.

It is a happy circumstance that locating the main program will also identify the memory model used. Since the C main() function has three parameters (argc, argv, and envp), the call to main will almost always be compiled to three (or more) push instructions, then a call. In the case of the second and third parameters, these are pointers to data, and so 16 bits will be pushed in the case of small and medium models, and 32 bits will be pushed in the case of compact and large models. Similarly, the call to main itself will be either a near call (small and compact models), or far (medium or large models). Tiny model programs (those with a .com extension) have slightly different startup code, but have small model library functions. *dcc* checks for the tiny model startup code, and if found, treats the program as if it were small model. Turbo Pascal® programs are identified by the far call to the program initialisation code as the very first instruction. Version identification is made from the signature of the program initialisation code.

It may be desirable in the future to locate these signatures outside of *dcc* in some kind of directory file (in the sense of a paper directory). This file would contain signatures and file names, and could be maintained and added to without having to recompile *dcc*.

Since the number of signature files could be large (several vendors, several versions of library files, and several models), all signature files can be located in a special directory, pointed to by an environment variable.

8 Results

One signature file is generated automatically from the appropriate library file in a few seconds, with no user intervention required. Other techniques for recognising signatures [6] may require a large effort on the part of the user, and this effort has to be largely repeated for each new vendor, model, and version combination. The resultant signature files occupy approximately 40 to 80 kilobytes of storage. Most commonly, the mapping phase of the algorithm is executed only once or twice. However, instances of 6 iterations have occasionally been observed.

The number of duplicate keys varies with the compiler vendor and version number from as low as 19 (of 357) for Turbo C® 2.01, to 440 (of 1347) for Microsoft® Visual C++™ V1.00. In the former case (Turbo C), most duplicate keys are due to identical implementations of functions with different names, e.g. spawnvp, spawnvpe, spawnve, spawnv, spawnlp, and spawnl. A few are related but not separable, due to the size of the signatures; e.g., tolower and toupper; lock and unlock. Only one pair occured because unrelated functions were too similar: brk and atoi. In the case of Microsoft Visual C++, most duplicate keys are due to internal public names that are not accessable by the user, for example _CIcosh and _CIfabs. These names are all 3 byte jumps. Since unconditional jumps terminate signatures, they are not distinguishable. Others are "decorated" C++ names, e.g. "??1strstream@@U" and "??1stdiostream@". Unfortunately, many of the trancendental maths functions are just mov dx,xxxx followed by unconditional jumps, so they are lost. Examples: cosl and coshl. As with the Turbo C case, there are some identical functions with different names, like _execv and _execvp. A few more are caused by Microsoft's naming convention: _fabs and _aNfabs. The latter is the near data version of the general _fabs function; in small model programs, the two are synonymous. (In large model programs, _fabs and _aFfabs are synonyms).

When these anomalies are taken into account, only a very small proportion of useful signatures are lost.

Signatures have been generated for Microsoft® C 5.1, Microsoft® Visual C++™ V1.00, Turbo C® 2.01, Borland C V3.0, Turbo Pascal® version 4.0 and 5.0, and Logitech Modula V1.0.

9 Prototypes

Not all functions found in the library files are normal library funtions; some are compiler helper routines like LXMUL@ (long integer multiplication). These do not have to follow high level calling conventions, and so it is useful for *dcc* to analyse these functions (e.g. to find which registers contain parameters on entry, or return values on exit). One source for differentiating normal library functions from others is the set of include files that comes with any C compiler. These contain prototypes for all callable library functions, but not the helper functions.

A program, called parsehdr, was written to parse header files, isolate prototypes, and save information about parameter and return types into a file. This file is used by *dcc* to determine whether or not to decompile a function. If the compiler vendor, model and version is not recognised, or if the appropriate signature file does not exist, all functions are decompiled. This provides information to *dcc* as to the type of variables in *dcc's* symbol table, and can help determine how to handle certain instructions. For example, in the case of

```
MOV   AX,42
CALL  printf
```

it is known from the prototype file that the parameter is a pointer to a char, which means that the operand (42) is actually an offset to a data item. When the data at offset 42 is examined as a character string, all text data to

the terminating null can be marked as data, and a meaningful label can be assigned to the offset 42. Therefore the disassembly can be improved to

```
MOV   AX, OFFSET szHelloWorld
```

and the decompiled

```
printf(0x42)
```

can be improved to

```
printf("Hello world\n").
```

In order to find the prototypes in the include files, a parser similar in many respects to that found in a compiler was used. For simplicity, #define and typedef statements were ignored. A few simple heuristics were introduced, to improve the proportion of correctly handled prototypes. These include "size_t" and "FILE" as unsigned int and pointer to struct respectively. All functions in the Turbo C 2.01 include directory that are associated with ordinary C functions (excluding the graphics library, for example) were processed successfully, apart from a very few with exotic types (like jmp_buf), and prototypes involving pointers to functions.

Turbo Pascal® prototypes are stored in the turbo.tpl library file. It would be possible to generate the prototype information for the Pascal signatures so that *dcc* could take advantage of parameter and return type information. For example, this would allow the parameter to WriteString to be recognised as a string, so that the Pascal statement

```
writeln("Hello Pascal");
```

would be improved in the *dcc* output file from

```
WriteString(0x42);
CrLf();
```

to

```
WriteString("Hello Pascal");
CrLf();
```

10 Conclusions

The use of signatures to identify library signatures has proved very helpful for *dcc*. The clarity of output is much improved, the volume of output is reduced, and the time spent performing analysis and decompilation of library routines is eliminated. Also, any unusual or difficult constructs that are encountered are now much more likely to be part of the oroginal program, and not some low level assembler library function. Therefore it is possible to call an interactive disassembler for those cases, and it is hoped that the expert user may be able to guide the analysis in the future, where required.

Not all library functions are recognised, but the vast majority are, and this is sufficient for the purposes of *dcc*. It is a general method, that works for many languages, compiler vendors, and memory models, and is completely automatic. The complete lack of user interaction compensates for the small number of library functions that are not recognised. Generation of the library files is quick and also completely automatic, given the appropriate library files. The signature files occupy only a moderate 50k each of disk space on average.

Acknowledgement

This research is funded by Australian Research Council (ARC) grant no. A4913061. Many thanks are due to Professors Bill Caelli and John Gough for their guidance.

References

[1] Z.J. Czech, G. Havas and B.S. Majewski. "An optimal algorithm for generating minimal perfect hash functions", *Information Processing Letters*, 43(5):257-264, October 1992. Also Technical Report TR0217, Key Centre for Software Technology, Department of Computer Science, University of Queensland 4072 Australia.

[2] G. Havas and B.S. Majewski. "Optimal algorithms for minimal perfect hashing", Technical Report TR0234, Key Centre for Software Technology, Department of Computer Science, University of Queensland 4072 Australia.

[3] B.S. Majewski, N.C. Wormald, G. Havas and Z.J. Czech. "A Family of Minimal Perfect Hash Functions", Technical Report TR0242, Key Centre for Software Technology, Department of Computer Science, University of Queensland 4072 Australia.

[4] G. Havas, B.S. Majewski, N.C. Wormald, and Z.J. Czech. "Graphs, Hypergraphs and Hashing", Technical Report TR0273, Key Centre for Software Technology, Department of Computer Science, University of Queensland 4072 Australia.

[5] Jürgen Ebert. "A versatile data structure for edge-oriented graph algorithms", *Comm ACM 30,6* (June 1987), 513-519.

[6] Chen Fuan, Lin Zangtian and Li Li. "Design and Implementation Techniques of the 8086 C Decompiling System", *Mini-Micro Syst. 14,4* (1993), pp.10-18, 31. Written in Chinese.

[7] W. L. Peavy. "Inside Turbo Pascal 6.0 Units", Public domain software file tpu6doc.txt in tpu6.zip. Anonymous ftp from garbo.uwasa.fi and mirrors, directory /pc/turbopas, 1991.