Identifying Library Functions in Executable File Using Patterns

Mike Van Emmerik
Center for Software Maintenance
Department of Computer Science and Electrical Engineering
The University of Queensland, Australia
emmerik@csee.uq.edu.au

Abstract

Re-engineering from legacy executable (binary) files is greatly facilitated by identifying and naming statically linked library functions. This paper presents an efficient method for generating files of patterns; each pattern is a transformation of the first several bytes of a library function's executable code. Given a suitable pattern file, a candidate function can be identified in linear time. One pattern file is generated for each combination of compiler vendor, version, and memory model (where applicable). The process of identifying these parameters in a given executable file also identifies the `main` *function of the program, i.e. the start of the code written by the user. The pattern files are produced automatically from a compiler's library file in a few seconds, with no user intervention required. Due to various limitations, not all library functions can be identified correctly; a small number will be either incorrectly identified or not identified. Optimal perfect hash functions are used to keep the pattern files compact and efficient to process.*

1. Introduction

A large part of the software engineering effort for any software product is spent after delivery. Much of this effort is spent maintaining the product, but often it is necessary to re-engineer legacy products to extend their lifetimes. An increasing number of re-engineering tools operate directly on executable files; examples include an executable editing library EEL [9], and a binary profiler TracePoint [10]. In some cases, the source code is not available any more, or it is desirable to prove that the executable file corresponds to a given source file set. Furthermore, for year 2000 testing, certain library functions will be known to contain year 2000 date bugs, while the vast majority will not be concerned with dates at all and can therefore be ignored. Also, the calling of date related library functions from application code will highlight areas of a program that deserve special attention.

When dealing with executable files, it is very useful to know whether any given function is a standard library function (and if so, what its name is), or part of the application. Identifying dynamically linked library functions is trivial, however most *legacy* programs use static linking. When decompiling an executable program, knowledge of the `main` function and the names of library functions are very useful, since library code is much more difficult to decompile, and often it is not desirable to decompile the library functions at all. In binary translation, there are often native library functions available with the same name and semantics in the target system, so identification of library functions is very useful. Other applications include debuggers, optimisers, code improvers, and profilers.

When a function is presented for identification, a pattern is made by transforming the first n bytes of its executable image. The reason for the transformation will be given shortly. It is then compared with the patterns for all library functions that could have been statically linked with the executable file. These patterns are stored in a pattern file, and are derived from the same library file that was used to link the executable file. Hashing is used to efficiently locate the pattern for the candidate function in the table of patterns for all library functions. If found, the name of the function is found by indexing a table of function names, also derived from the original library file.

For simplicity, n is chosen to be fixed, even though it turns out that this decision implies that some library

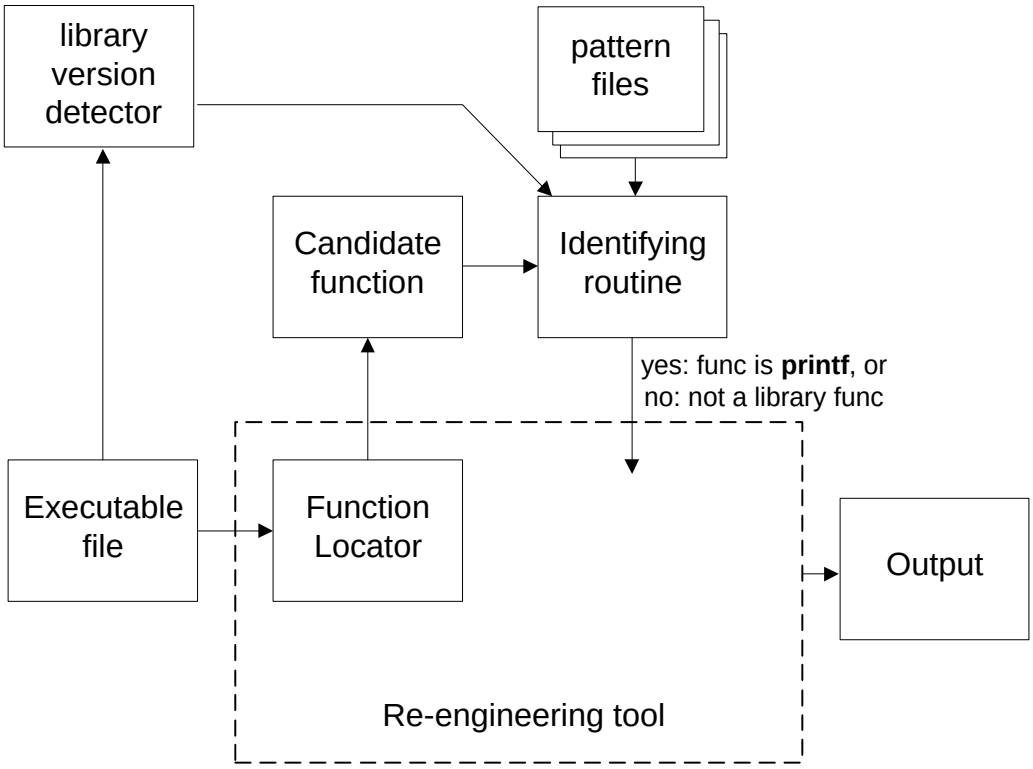

Figure 1. Overall diagram of a re-engineering tool using patterns

functions won't be able to be identified. The transformation mentioned above is required to solve two problems. Firstly, relocation changes bits in a function's executable image. Secondly, for a sufficiently large value of n that can separate similar functions, there will always be functions short enough that code which is not part of the function will corrupt the pattern, causing the pattern comparison to fail needlessly.

Optimal perfect hash functions are an ideal way to look up a candidate function's pattern in a table of fixed patterns. However, the lack of unique keys presents problems with the standard algorithm for generating such hash functions.

This paper discusses how these problems were overcome while developing *dcc* [7], an experimental decompiler.[1] It is hoped that writers of other re-engineering tools that operate on executable files will be able to adapt the techniques of this paper, so their tools will be able to identify library functions. The *dcc* decompiler translates Intel 80286 executable files for the MS-DOS operating system to the C language. The techniques would, however, be applicable to any architecture and operating system that uses static linking of library functions, with minor changes.

1.1. Pattern Files

Figure 1 is an overall diagram of a re-engineering tool using patterns. Patterns are used by the Identifying Routine to decide whether a canditate function is a library function, or not. A pattern is a transformation of the first n bytes of the library function's executable code. The transformations are wildcarding and truncation, and are discussed in Sections 3 and 4 respectively. A pattern file is generated once; one is needed for each library file. When a re-engineering tool examines an executable file, it attempts to detect enough information to decide which pattern file is required.

For example, it might be found that the executable file was linked with Borland C version 2 for the compact memory model. If the pattern file is found in the appropriate path, the tool attempts to match every function of interest with the patterns in the pattern file.

[1] Patterns were previously known as signatures in the *dcc* literature.

Hashing is used so that effectively one pattern (from a candidate function) is compared with all patterns in the pattern file at once. If there is a match, then the name of the function is known; if not, the function is not a library function. The tool can use this information to prevent further analysis of the function (e.g. if it is a decompiler), or initiate special analysis of the function (e.g. if the function is known to be of interest for year 2000 analysis).

The pattern file would have been generated by running the pattern generator tool, using the appropriate library file (e.g. `cc.lib`) for a particular compiler.

Also stored in the pattern file are tables that define a hash function which is the essential part of the identifying routine.

2. Library Files

A library file is an object file supplied by a compiler vendor to be linked with the user's program. This file is ideal for generating the pattern files, since it contains the executable code for all runtime library functions, and their names. MS-DOS is an example of a legacy platform whose executables all use static linking. Vendors for MS-DOS compilers usually provide several library files; one for each memory model supported, and perhaps one for each choice of floating point support (often direct, emulated, and alternate).

The particular library function image that will be found in an executable file will depend on the combination of vendor, version, and memory model. This variation is overcome by using a different pattern file for each supported combination. A simple naming convention identifies the required pattern file, once the vendor, version, and memory model are found. The user need generate only those pattern files that s/he is likely to need.

Some compilers do not use standard library (.lib) files for library code. In this case, custom programs have to be written to generate the pattern file from the equivalent of a `.lib` file. Such a program has been written to read a Turbo Pascal `turbo.tpu` file (the equivalent of a library file) and generate a pattern file.

3. Wildcards

Instructions in executable files have parts that do not change with code or data position, and parts that are position dependent. Some parts may be dependent on the position of the code; others on the position of the data. The source of the dependancy is not important, since both will cause the final binary bit pattern to be unsuitable for pattern matching. Position independent parts of the instruction include opcodes, program relative operands, indexes to the stack frame, and some immediate operands. For example in the 80286 instruction

```
8A 46 42    mov ax, [bp+42]
```

the opcode (8A), the mod/rm byte (46) and the stack 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 or may not be invariant, since it might represent an offset into the data segment, or may represent a genuine constant. In the pattern files, position dependent bytes are replaced with a special wildcard byte, 0xF4 (the opcode for the `halt` instruction). So the instruction above would be stored within a pattern file as

```
B8 F4 F4
```

Pattern files contain essentially the first n bytes of a library function's compiled instructions, with variable bytes replaced by the wildcard. The value for n is important. If n is chosen too small, then some library functions will not be distinguishable. If n is chosen too large, then as well as taking up more space in the pattern files, there is more chance of the pattern running into either another library function, or unrelated code. (There is protection against this, as discussed in Section 4). Experimentally, for 80286 MS-DOS code, 23 bytes was found to be a good value for n. Almost all functions have a 3 byte sequence to initialise the stack frame, leaving 20 bytes to differentiate one library function from others.

4. Truncating the patterns

It is important to ensure that the pattern for a particular library function does not extend beyond the end of the function, since whatever comes after the end of the function won't ever match (see Figure 2). Some criteria must be used to determine the end of the function. The same criteria must be used to truncate the pattern when it is generated, and when a candidate function is passed to the identifying routine, so that the resultant patterns will match when required.

An obvious criterion is to truncate the pattern at the actual end of the library function, but unfortunately finding the end of a function is undecidable. However, it was found that a simple heuristic works

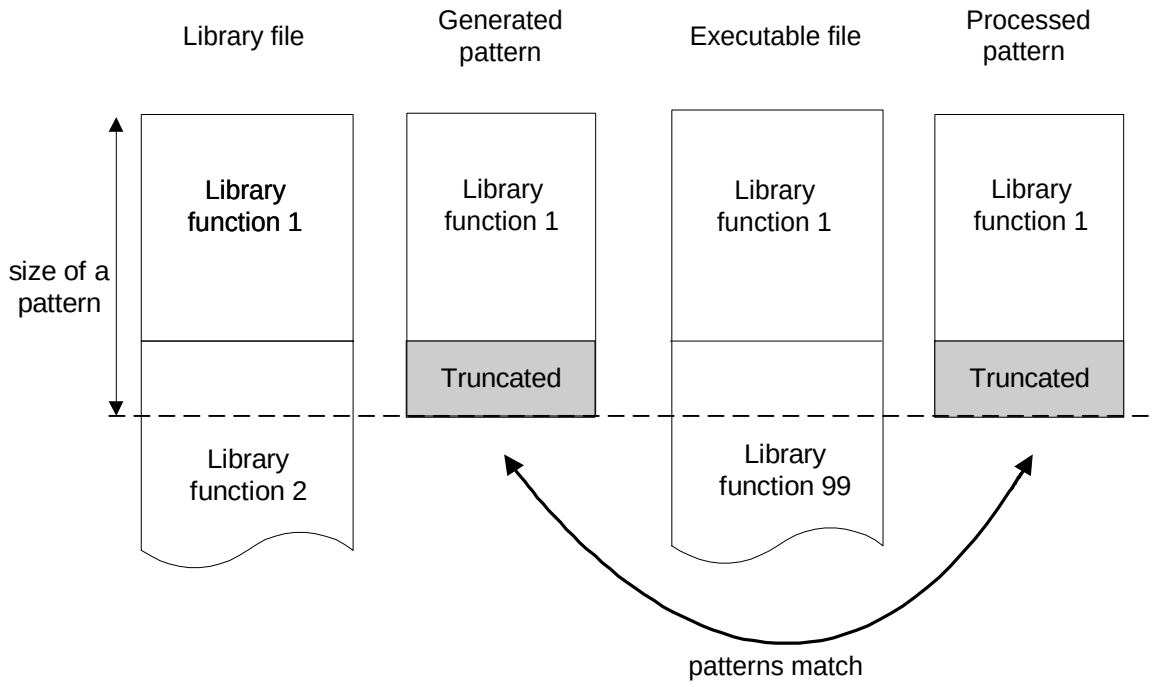

Figure 2. Truncation of patterns

well. The heuristic is to truncate after the first unconditional branch, indirect jump (register indirect, memory indirect, or both), or subroutine return instruction. This heuristic also solves the problem of undecidability when disassembling the library function (disassembling is required for inserting wildcards, and for finding the first instruction that will truncate the pattern). Patterns are truncated by inserting binary zeros up to the end of the pattern.

This heuristic brings great simplification to the pattern generating and comparison process, but it it is also the factor that contributes most to unidentified functions (see Section 7 for how well the fixed patterns are able to recognise library functions in practice).

Figure 3 depicts a typical pattern. Only the second column is actually part of the pattern (i.e. the hexadecimal values 55, 8b, ec,...); the other columns are for annotation only.

```
0100 55       push bp
0101 8bec     mov bp,sp
0103 56       push si
0104 8b7604   mov si,[bp+4]
0107 56       push si
              ;Destination wildcarded:
0108 e8F4F4   call XXXX
010b 59       pop cx
010c 0bc0     or ax,ax
010e 7405     jz 0115
              ;Operand wildcarded:
0110 b8F4F4   mov ax,xxxx
              ;First uncond branch:
0113 eb4c     jmp  0161
0115 0000     ;Padding (truncation)
```

Figure 3. Pattern for library function `fseek`

5. Hashing the Patterns

The detailed workings of the algorithm will not be given here, as it is available in the literature [1, 2]. However, a very broad outline will be given, to explain a required modification to the standard algorithm. Logically, an optimal perfect hash function is generated from the array of patterns. In practice, the hash function is fixed, and takes as parameters two random tables, and a third table generated from the two random tables. The mapping phase of the algorithm (see Figure 4) initialises the two tables literally with random values (within a certain range).

Not all sets of random numbers are suitable; this is the penalty of randomised functions. When the number set is not suitable, it is guaranteed that a graph

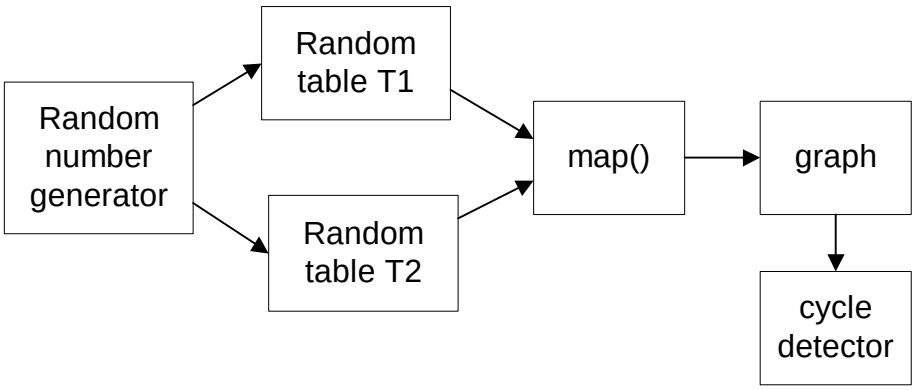

Figure 4. The mapping phase of the algorithm that generates a tailored hash function for a given set of patterns. This phase generates tables of random numbers, derives a graph from these tables, and tests the graph for cycles. If a cycle is found, the tables are generated again with different random numbers.

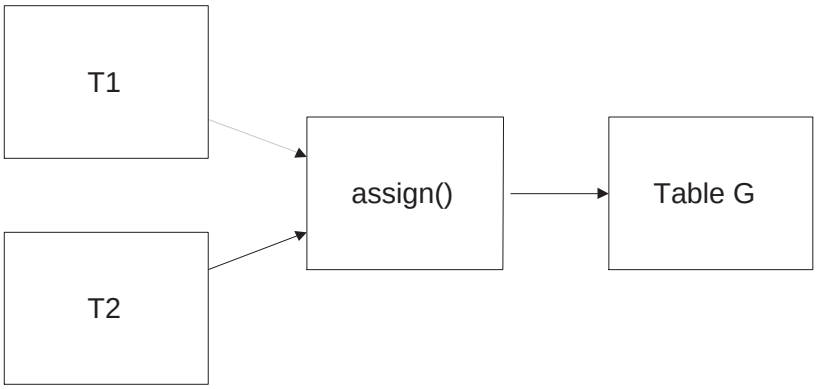

Figure 5. The assignment phase of the algorithm that generates a tailored hash function for a given set of patterns. This phase generates the table "g" from the random (but now validated) tables T1 and T2.

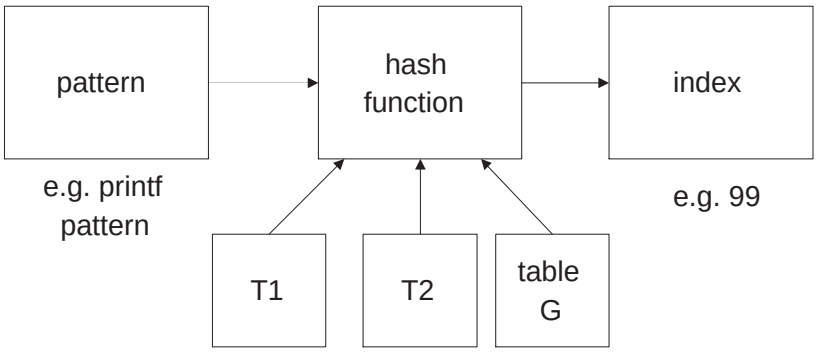

Figure 6. The hash function is a fixed routine that uses three tables as parameters, as well as the input key (a pattern derived from the candidate function). The output is an index (0 for the first pattern in the table, 1 for the second pattern, etc).

associated with the set contains a cycle; suitable sets guarantee an acyclic graph. This graph is tested for cycles, and if one is found (i.e. the set of random numbers is unsuitable), new random numbers are generated, and the cycle is repeated. The size of the random tables is such that there is a good chance that suitable tables will be found in a few iterations of this cycle; hence the generation is said to operate in linear random time.

After the mapping phase succeeds, the assignment phase (see Figure 5) generates another table (table g), which is used by the hash function (see Figure 6) to ultimately decide whether the presented pattern (from the candidate function) represents a library function or not. If the pattern presented does not match any key, the hash function will still always generate a valid index, so a comparison of the indexed pattern with the input pattern is always required.

A table based representation [4] of the graph was found to be convenient and compact.

Perfect (collision free) optimal (100% full) hash functions [1, 2] are a good way of storing the patterns, since the keys are constant. The keys of the hash function are the patterns. The implementation involves randomised algorithms[2], and generates the tables required for the hash function in random linear time. The hash function itself (and hence, the identifying routine) executes in linear time, being little more than two standard hash functions.

5.1. Duplicate Keys

It is an assumption of perfect hash functions that the input keys are unique; a hash function cannot return unique values when the input keys are not unique. However, not all library function patterns will be unique, for various reasons. One reason is that some library functions are themselves not unique, e.g. `unlink()` and `remove()` on MS-DOS machines are the same, since MS-DOS does not have Unix style links to files (therefore, removing a link to a file is the same as removing the file). However the main reason for the duplicates is that some library functions are just not unique in the first n bytes, especially after wildcarding and truncation.

Regardless of the values in the tables, two keys with the same value will always imply the same pair of vertices in the graph, and the edges will always be distinct. This will guarantee a cycle in the graph involving two edges, so the mapping phase (generating the tables) will never terminate. The required modification to the algorithm is as follows (see Figure 7). Where a cycle is detected, and more than one edge is involved, the keys associated with the two edges are compared. If the keys are different, then this is a genuine cycle, and the tables are unsuitable, as before. However, if the keys are the same, then this cycle is due to the duplicate keys, not a genuine cycle, and so the tables are not rejected.

[2]Randomised algorithms use tables of random numbers, and other table(s) computed from those random tables, to compute their results. To those unfamiliar with these algorithms, the results can appear quite magical. Interested readers can read [3] for an introduction.

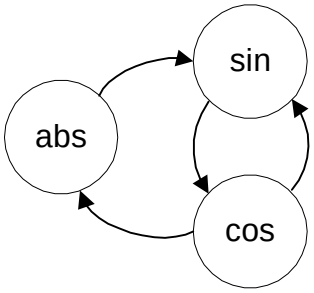

Figure 7. An example graph

In the example graph of Figure 7, there is a cycle caused by the library patterns `sin` and `cos` not being distinguishable (i.e. the patterns are the same). Even when this cycle is eliminated from consideration, it turns out that there is a cycle involving all three vertices, so the mapping phase of the algorithm to generate the hash function's random tables must be run again.

6. Choosing the Pattern File

When the re-engineering tool first examines an executable file, it has to determine which pattern file (if any) contains the patterns matching the library functions in it. This is done by comparing the first few hundred bytes of the executable file's code against a small set of fixed patterns. For C programs, the actual call to the `main` function is ideal for this, since the pushing of `argc`, `argv`, and `envp` (which happens even if some or all of these are not used) will determine whether near or far data is used, and the call to `main` itself will indicate whether references to code are near or far. These two pieces of information determine which of 4 memory models are in use, as well as the actual address of `main`. Other short patterns (of about 20 bytes) are used to determine the vendor and version of the compiler used.

7. Results

Each pattern file is generated automatically from a C library file using the `makedsig` tool in a few seconds. No user intervention is required; with other methods, much manual effort is typically required (e.g. see [5]). Another tool, `makedstp`, generates pattern files from the Turbo Pascal `turbo.tpu` file. Pattern files occupy approximately 40Kb to 100Kb of storage each. Most commonly, the mapping phase is executed once or twice, although instances of 6 or more iterations have occasionally been observed.

The proportion of duplicate keys varies from 5% for Turbo C 2.01, to 35% for Microsoft Visual C++ 1.00. In the former case, most duplicate keys are due to identical implementations of functions with different names, e.g. a set of six functions related to `spawnvp`. A few duplicates are due to related functions that are not separable because of the size of the patterns, e.g. `toupper` and `tolower`. Only one duplicate is due to unrelated functions that happened to have code that is too similar in the first 23 bytes: `brk` and `atoi`. In the case of Microsoft Visual C++, most duplicates are due to internal public names that are not accessable by the user, e.g. `_CIcosh` and `_CIfabs`. These library functions begin with absolute jumps, and so the pattern for them all is the same. Unfortunately, many of the transcendental maths functions are just a `mov dx,xxxx` instruction followed by a jump, so they are not distinguishable. Examples include `cosl` and `coshl`. A few more are caused by Microsoft's naming convention: `_fabs` and `_aFfabs`. The latter is the near data version of the general `_fabs` function; in small and medium model programs, the two names point to the same code. When these anomalies are taken into account, only a small proportion (a few percent) of useful library functions are not recognised.

It would seem that an extension of this system that works on variable length patterns would overcome the duplicate pattern problem. Indeed, this appears to have been achieved with the commercial disassembler IDA Pro [8].

Pattern files have been generated for Microsoft C 5.1, Microsoft Visual C++ V1.00, Turbo C 2.01, Borland C V3.0, Turbo Pascal versions 4.0 and 5.0, and Logitech Modula V1.0. They were first used with *dcc* [7], an experimental decompiler. Patterns helped *dcc* considerably, firstly by not attempting to decompile irrelevant parts of the input executable file, and secondly by giving the correct names to library functions. An additional advantage is that library code is typically very difficult to decompile, compared to ordinary user code from a compiler. When combined with a system that found parameter and return type information, *dcc* was able to generate code very similar to the original C source for simple test files, except for variable names (which are not stored in any symbol table in the executable file).

8. Conclusions

The use of patterns for identifying library functions in legacy programs has proved to be efficient and beneficial to the *dcc* re-engineering tool. Other such tools, such as those locating or correcting year 2000 problems, would also benefit from the use of patterns. Transformations are needed to prevent position dependent parts of instructions from failing pattern comparisons. Other transformations are necessary to cater for short patterns. Perfect optimal hash functions, with one minor modification to the standard algorithm, are an ideal way to index the fixed length binary patterns. Fixed length patterns do however have the limitation that some library functions cannot be distinguished from others. No manual intervention is required to generate the pattern files; all that is needed is the library file (.lib or equivalent) for the compiler used with the executable file. The tools automatically detect which of several pattern files to use for a particular executable file.

Where there are duplicate patterns, special care has to be taken when indexing the patterns with perfect optimal hash functions. Fortunately, there is a simple modification to the standard algorithm that allows suitable hash functions to be generated.

Acknowledgments

This research was funded by the Australian Research Council (ARC grant no. A4913061). Thanks to Cristina Cifuentes and Norman Ramsey for suggestions on improving the presentation of this paper.

References

[1] Z.J. Czech, G. Havas, and B.S. Majewski, "Perfect Hashing", *Theoretical Computer Science*, No. 182, 1997, pp. 1-143.

[2] Z.J. Czech, G. Havas, and B.S. Majewski, "An optimal algorithm for generating minimal perfect hash functions", *Information Processing Letters*, Vol. 43, No. 5, October 1992, pp. 257-264. Also Technical Report TR0217, Department of Computer Science, University of Queensland 4072 Australia.

[3] R. Gupta, S. Bhaskar, and S. Smolka, "On Randomization in Sequential and Distributed Algorithms", *ACM comput. Surveys*, Vol. 26, No. 1, March 1994, pp. 7-86.

[4] Jürgen Ebert, "A versatile data structure for edge-oriented graph algorithms", *Communications of the ACM*, Vol. 30, No. 6, June 1987, pp. 513-519.

[5] Chen Fuan, Lin Zangtian, and Li Li, "Design and Implementation Techniques of the 8086 C Decompiling System", *Mini-Micro Systems*, Vol. 14, No. 4, 1993, pp. 10-18, 31. Written in Chinese.

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

[7] C. Cifuentes and K.J. Gough, "Decompilation of binary programs", *Software – Practice and Experience*, Vol. 25, No. 7, 1995, pp. 811-829.

[8] Ilfak Guilfanov, "FLIRT (Fast Library Identification and Recognition Technology)", URL: `http://www.datarescue.com/idaflirt.htm`

[9] James Larus, "EEL: An Executable Editing Library", URL: `http://www.cs.wisc.edu/~larus/eel.html`

[10] "Tracepoint, An execution profiler for Windows/NT binary programs", URL: `http://www.tracepoint.com`