Using a Decompiler for Real-World Source Recovery

Mike Van Emmerik and Trent Waddington
School of ITEE, University of Queensland
Brisbane QLD 4072, Australia
{emmerik, trent}@itee.uq.edu.au

Abstract

Despite their 40 year history, native executable decompilers have found very limited practical application in commercial projects. The success of Java decompilers is well known, and a few decompilers perform well by recognising patterns from specific compilers.

This paper describes the experience gained from applying a native executable decompiler, assisted by a commercial disassembler and hand editing, to a real-world Windows-based application. The clients had source code for a prototype version of the program, and an executable that performed better, for which the source code was not available. The project was to recover the algorithm at the core of the program, and if time permitted, the recovery of other pieces of source code.

Despite the difficulties, the core algorithm was successfully decompiled, and a portion of the rest of the program as well. There were surprises, including the ability to recover almost all original class names, and the complete class hierarchy.

Keywords: Reverse engineering, decompilation, source code recovery, native executable file, experience.

1. Introduction

Decompilation researchers are regularly asked if they can recover lost source code for various native executable files. To date, the answer has always been the same: general decompilation is not a mature technology, there will be some chance with a good disassembler, and otherwise they have no realistic alternative to rewriting. Around August 2003, however, one potential source code recovery project seemed much more tractable. Their application was a Windows-based executable for speech analysis with heavy mathematics processing, and they had source code to a prototype version of the program. The clients were investors who purchased the rights to the program knowing that it needed further development. There was no time pressure. Importantly, they were not interested in decompiling the entire application; they were mainly interested in the core algorithm, and were intending to rewrite the user interface. The version for which they had source code compiled and ran, but the results were not as repeatable or as reliable as those of the final version, for which they did not have source code. The goal was therefore to provide source code for a program that provided the same results as the supplied executable program, using the prototype source code as a base.

The clients were told that there would be no guarantees. They would be contributing to decompilation research as well as having their algorithm recovered. They agreed to this; despite the potential problems, the only alternative was to accept the lower reliability of the prototype version. Rewriting was not an option, since the original authors were not available, and the final algorithms were not documented in any detail.

If all went well, the clients were also interested in some aspects of the main program, for example, functions that displayed the results in various graphs.

This is an experience paper reporting the decompilation of a real-world Windows-based application. Such a project is rarely attempted, and seldom reported in the literature.

The remainder of the paper is structured as follows. Section 2 provides a background on decompilers and how the decompiler Boomerang, which is currently under development, was used in this project. Section 3 lists the various problems that were encountered, and the solutions used to overcome them. The important issue of testing the decompiled code is discussed in Section 4. The results and lessons learned are summarised in Sections 5 and 6. Issues that remain for future work are discussed in Section 7, while Section 8 concludes the paper.

2. Decompilation

A decompiler is a reverse engineering tool that takes as input a program in the form of an executable file, and produces a high level language representation of that program [7]. For the purposes of this paper, the executable file will contain native machine code, although decompilers exist for Java bytecodes, Visual Basic, and so on. Decompilers find application in software security, maintenance, interoperability, verification, and more. While Java decompilers are largely successful, general native executable decompilers so far rarely generate a correct, readable, high-level representation of the program.

Decompilation has a surprisingly long history, going back to the early 1960s [7]. While general decompilers are immature, pattern based decompilers [12], which are tied to a particular compiler, can be commercially successful (e.g. [8, 14]). Source Recovery [13] have a commercial C++ decompiler for Hewlett-Packard native executables.

One of the first decompilers to use general techniques was *dcc* [4, 3, 5]. Dcc is limited to the 80286 platform, whereas REC can decompile programs compiled for a variety of platforms [2]. However, source code for REC is not publicly available.

Native decompilers are more successful when starting from assembly language. Mycroft translated legacy BCPL programs into an assembly-like language before decompilation [11], while Ward was able to decompile mainframe and 80186 assembler to C in an industrial setting [17, 18].

2.1. The Decompilation Process

Figure 1 shows how decompilation can bridge the gap in the edit-compile cycle of a program.

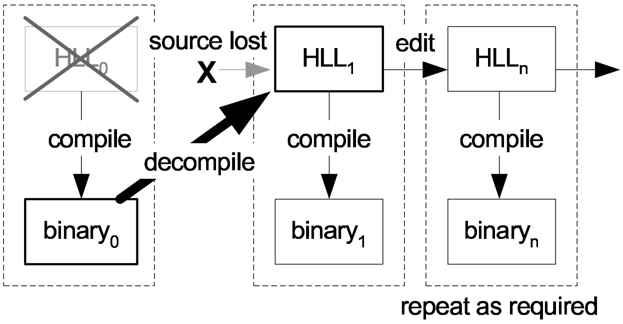

Figure 1: Decompilation bridges the "edit-compile cycle" break.

The process of decompilation for this project was as follows. A commercial disassembler, IDA Pro [6], was used to explore the executable file. Using the disassembler, various procedures of interest could be identified. For example, the procedures exported by the Dynamically Linked Libraries (DLLs) were immediately of interest, and with some effort, it was possible to find the procedure that is called when key user interface elements were activated (toolbar buttons, menu commands, etc.) Once a procedure was identified, it was entered into a special symbol file; Section 3.2 has more. It was given a suitable name, and its parameters were named and typed. One procedure was focused on at a time.

With this preparation, the Boomerang decompiler (Section 2.2) was run. The decompiler may have crashed for various reasons. If this happened, it was debugged, and either the bug was fixed, other command line switches were passed to it, or new switches were added to the decompiler to work around the problem.

Once some output was obtained, it was inspected. In every case, there were obvious problems with the decompiled output. Where practical, these were corrected by modifying Boomerang. Otherwise, the output was edited to make the code correct, or more readable. For example, certain idiomatic sequences of instructions (Section 3.6) had to be replaced with known equivalents in C code.

At this stage, it may have been possible to compare the code against the corresponding prototype source code (if any). It was often possible to rename generic variable names (e.g. `local6`) to meaningful names (e.g. `tickSize.cx`). As understanding of the code gradually increased with this process, it was usually possible to discover the type of parameters to the current procedure. If it was a pointer to a structure which had a close equivalent in the prototype source code, then structure members could be named. It was firstly named in the disassembler, then the structure definition was copied into the symbol file, and the decompilation of this procedure was repeated. It was important to take structure definitions from the disassembler, not from the prototype source code, because of the likelihood that a member variable could have been added, removed, or moved since the prototype was written.

The process was repeated until all procedures of interest were decompiled. At that stage, the code had to be tested (Section 4). Often the tested code would fail, because some variable was not initialised (in the code that had been decompiled to that point). Searching for where a memory location was initialised was a frequent problem.

2.2. Boomerang

Boomerang is an attempt at a general, open source, retargetable decompiler of native executable files [1]. While a compiler has a machine dependent back end, a decompiler has a machine dependent front end. The loader and decoder (main parts of the front end) are said to be source machine dependent. Boomerang has several front ends, translating the instruction stream into an intermediate representation called Register Transfer Language (RTL). A machine independent analysis engine transforms the RTL into high level form, and a back end emits the results in a high level language. Presently, there is only one back end, for the C language. Compiler techniques such as data-flow analysis (using the Static Single Assignment form) and control-flow analysis are used. Boomerang is able to decompile small test programs (e.g. calculating Fibonacci numbers), with no user intervention, to readable, compilable C. It handles recursion, recovers parameters and return values, switch statements, and so on. Larger programs provide more opportunity for things to go wrong in a decompilation; decompilation effort seems to increase at least linearly with the size of the input program. Clearly, Boomerang was not ready for general commercial use, and is still a long way from that stage.

3. Problems and Solutions

The following subsections discuss the problems that emerged during the project, and what steps were taken to either fix the problem, or work around it.

3.1. Program Size

The first problem was the sheer size of the program. The main 32-bit executable file was 670KB in size, with two DLLs of 138KB and 36KB. The prototype program was written in an early version of Microsoft Visual C++ (MSVC), and compiled to a main 16-bit executable file of about 250KB. Because of the technology differences, it is difficult to compare how much functionality was added between the prototype and the final version. Contrast these sizes with the test files that Boomerang was known to be able to handle, all of which were well under 20KB.

Source code for the prototype version gave the names of the procedures of interest. Fortunately, these happened to be exported by the larger DLL. Procedure names are usually only available in an executable file if the procedure is called by name, e.g. from the main program to a DLL. It was quickly determined there were about 8 procedures of major interest, each originally only a few pages of C source code.

3.2. Library Code

It is important to recognise library code which has been statically linked into the program for three main reasons: naming the library functions improves the readability of the decompiled code dramatically, it produces a wealth of type information, and library code is not part of the user's program. The compiler (of the decompiled code) will add library code automatically.

Library code (e.g. `printf`, `MoveTo`) can be recognised using a form of pattern matching [15]. This is currently missing from Boomerang; however technology similar to this is in the disassembler IDA Pro, where it is called FLIRT. This disassembler was used to identify the addresses of code to avoid decompiling.

Each procedure to be decompiled could have several library functions that it calls (which are not to be decompiled). This was one of the motivations for adding a command line option to read a symbol file. The symbol file can contain multiple lines of the form:

```
0x450260 __nodecode void CStringDestruct();
```

In the above line, the procedure starting at `0x450260` is given a name and a return type. The `__nodecode` modifier prevents decompilation. The present inability to specify classes (Section 7) is worked around with a naming convention. Most calls to destructors disappear in the final code anyway; presently they are deleted by hand.

For this project, recognising library functions did not eliminate very much of the bulk of the code section to be examined. The main executable was dynamically linked with library code; "library code" (mostly jump instructions) took up less than 5% of the total code section. The main DLL of interest was statically linked with library code; 80% of the code section was library code. Because of the closeness of this part of the code with the prototype source code, it was already known which 8 or so procedures were of major interest.

3.3. Entry Points

Decompilers can use a variety of approaches to the problem of separating code and data. For a good survey of these, see Section 2.1 of [16]. Boomerang uses data-flow guided Recursive Traversal: each procedure is decoded, then those it calls, starting with the `main` procedure. Locating `main` is source machine dependent. This approach avoids decompiling the runtime startup code, which like library code is not part of the user's source code, and is also inserted automatically by the compiler.

Windows-based programs, while containing a top level procedure called `winmain`, have hundreds of entry points, very few of which are reachable directly from `winmain`. Many of these are pointed to by operating system specific structures such as message maps (Section 6.1). Other entry points are callback procedures, which can be deduced from the parameters to library functions that take such parameters.

The temporary solution was to add command line switches so that a single procedure could be decompiled.

3.4. Types and Structure Members

Calls to library functions are a rich source of type information, since the types of the library functions' parameters (if any) and return type (if any) are published. This was not so much of a concern with the mathematically oriented DLL, since it dealt mainly with the types `float`, `double`, and `int`, together with arrays and structures of these. However, Graphical User Interface (GUI) code (such as in the main executable) deals with many types, such as points, bitmaps, brushes, fonts, etc. Real-world programs often contain extensive GUI code, and therefore rely on types, much more so than the text based test programs Boomerang had been tested with to date.

When this project started, Boomerang had very weak support for types. By the end, there was support for named structure members, which appear in the C output with the `.` and `->` operators. Type information from calls to library functions were also propagated to other variables. As an example, an expression could now resolve to `pView->printClientRect.right`. These two major updates to Boomerang improved readability dramatically, compared with the earlier `*(int*)(pView+56)`. Also added during the project was the ability to paste complete structure definitions from header files into the symbol file mentioned earlier. Figure 2 shows an example from a view class. Boomerang could now correctly type most parameters, local variables, and global variables.

```
typedef struct {
  CView vw;
  RECT  printClientRect;
  int   field_58[11];
  CFont dispFontVert;
  CFont printFontVert;
  int   showGraph[3];
 ...
} CLongPlotLine;
```

Figure 2: A class definition from a symbol file.

The example class (in the original source code) was derived from class `CView` (a built-in, abstract class). This is implemented (by this compiler and most others) by inserting a complete `CView` structure at the start of the `CLongPlotLine` structure. Hence, in the Boomerang structure file, the line `CView vw;` is prepended to the class' definition. Some fields whose purpose was not discovered (they were not present in the prototype source code) retain generic names such as `field_58`.

3.5. Floating Point Semantics

Mathematically oriented programs deal largely with floating point numbers. Floating point instructions were relatively uncommon in the programs Boomerang had been tested with to date. As a result, bugs were uncovered in the semantic specification of some floating point instructions. The Pentium processor has a particularly complex set of floating point instructions; the semantics of some instructions are so complex that several current disassemblers still output incorrect results. Early versions of Intel manuals also have errors (e.g. the description of FSUB ST(i), ST and FSUBR ST(i), ST [9, 10]).

The solution was found to be simple but laborious: check and recheck the semantics of each instruction (in the Semantics Specification Language (SSL) file) against the published specifications. In a few of the worst cases, it was found necessary to write an assembly language test program. This program was single stepped with a debugger in order to determine the exact semantics of an instruction.

3.6. Idioms

```
10002BA7 D9 EA  fldl2e          ; push log_2 e
10002BA9 DC 0D+ fmul ds:d690    ; push argument
10002BAF D9 C0  fld st
10002BB1 D9 FC  frndint
10002BB3 D9 C9  fxch st(1)
10002BB5 D8 E1  fsub st, st(1)
10002BB7 D9 F0  f2xm1
10002BB9 D9 E8  fld1
10002BBB DE C1  faddp st(1), st
10002BBD D9 FD  fscale
10002BBF DD D9  fstp st(1)
10002BC1 DD 1D+ fstp expSF      ; Store result
```

Figure 3: Assembly language sequence resulting from in-lining a call to the C exp(x) function.

Idiomatic instruction sequences (idioms) are sequences whose meaning is not obvious from the meaning of the individual instructions. The MSVC compiler used several idioms to implement certain pieces of high level code. For example, Figure 3 shows a sequence of 11 instructions implementing a call to the C library function `exp(x)` (raising *e* = 2.718... to the power of `x`). As with natural language idioms, "you just have to know" that this language element (here a pattern of instructions) has an associated meaning (here "raises *e* to the power of `x`").

```
40214F B8 AB+ mov eax, 2AAAAAABh ; 2^32/6
402158 F7 E9  imul ecx           ; edx = ecx/6
40215A 8B CA  mov ecx, edx
40215C C1 E9+ shr ecx, 31        ; -1 if result -ve
40215F 03 D1  add edx, ecx       ; adjust
402161 89 54+ mov [esp+...], edx ; store result
```

Figure 4: Using a multiply instruction to perform integer division by a constant.

Another common idiom was to use a multiply instruction to perform division by an integer constant. Figure 4 shows code for dividing register `ecx` by 6, and storing the result in a stack variable. The 32-bit multiply instruction produces a 64-bit result, with the high word in register `edx`. The adjustment implements the C requirement of truncation towards zero. For this project, such idiomatic sequences were replaced by hand.

3.7. Register Calling Convention

Unix calling conventions for the Pentium processor (e.g. Linux, Solaris/X86) do not pass procedure parameters in registers. Some MSVC procedures use the “thiscall” calling convention, where the first parameter is passed in machine register `ecx`. This fits particularly well with C++ class functions, which have an implicit `this` parameter, hence the name.

This was worked around with the `__custom` tag, e.g.

```
0x00450B00 __nodecode __custom int
  CWnd_ShowWindow(
    r[25]:  void* this,
    m[r[28] + 4]:  int bShow);
```

When `__custom` is used, all parameters are preceded by an expression representing the address of that parameter and a colon. In the example above, `m[r[28]+4]` is the standard location for the first argument (`r[28]` is the Pentium stack pointer).

Obviously, direct support for several of the more common register calling conventions would be advantageous. For example, it appears that Borland code commonly passes up to 3 parameters in registers.

4. Testing

Two kinds of testing are used in forward engineering: unit testing and functional testing. Unit testing tests a unit of code in isolation; functional testing tests the program as a whole. Pure unit testing is impractical in a decompilation situation, because there will in general not be enough understanding of each unit (a newly decompiled procedure) to write a test. However, each procedure can be tested with known working code (either original code, or a combination of original code and new but tested code), as explained in Section 4.1. Functional testing of a decompiled program amounts to comparing two programs. One is the original program; the other is the result of recompiling the decompiled program (the recompiled program). Comparing programs is not always straightforward, as discussed in Section 4.2.

4.1. Using DLLs

Once all the procedures of interest were decompiled from the mathematically oriented DLL, testing of those procedures was simple. A new DLL was compiled, and the original main executable was made to call the new DLL. This was accomplished by putting the new code in a suitable directory, and renaming the old DLL. The new code was compiled with debugging information.

In fact, the DLL mechanism is very handy for testing even a single procedure. The beginning of any decompiled procedure can be patched to make a call to a procedure in a DLL. It was found that this operation was so tedious and error prone that a tool was developed to automate the process.

Ultimately, however, it was found that once a related group of procedures was decompiled, it was compiled into a small stand-alone test program. Parts of this program started out with source code from the prototype code, suitably modified if necessary. Gradually, the prototype code was replaced until no prototype code (of interest) remained.

4.2. Comparing Programs

As mentioned above, final testing involved the comparison of the outputs from running two executable programs (the original and the recompiled programs). However, when this was done, some results were not identical to the fifth decimal place. It was important to decide whether this was an error in the decompilation, or merely a detail of the final recompilation. Initially, it seemed that the latter was true; different compilers in general do produce slightly different results for the same floating point source code. However, it became difficult to make that decision. For example, the recompiled program ended up producing close results for 19 out of 20 outputs, but the last one was very different. This seemed to indicate a decompilation error.

It was finally decided that the only way to be sure was to make modifications to the decompiled program, attempting to force the compiler to generate code more similar to the original executable file, so that the results would literally compare to the last decimal place. Figure 5 shows a fragment of original C source code which illustrates the process.

```
static float K[30];
static float* R;
double alpha; ...
K[0] = -R[1]/R[0];
alpha = alpha - K[0]*K[0]*alpha;
```

Figure 5: Original C code fragment.

Part of the disassembly for this code fragment is shown in Figure 6. Note that the top of the floating point stack is stored with 80 bits of precision, while the variable K is stored with 32 bits of precision. The multiplication is of the 80-bit version of `K[0]` with a 32-bit truncation of itself. To force the compilation to generate identical results, the decompiled output had to be modified as shown in Figure 7.

```
26BF D9 E0  fchs          ; ST = -R[1]/R[0]
26C1 D9 15+ fst K         ; K[0] = -R[1]/R[0]
26C7 D8 0D+ fmul K        ; K[0] * K[0]
```

Figure 6: Original machine code for the same fragment.

```
double temp1 = -R[1]/R[0];   // For repeatability
K[0] = (float)temp1;
alpha = alpha - temp1 * K[0] * alpha;
```

Figure 7: Modified output for the same fragment.

When several such changes had been made to the code, the two programs matched completely, using several different input files. Therefore, the unusual behaviour of the twentieth output, i.e. sensitively depending on the precision of intermediate results, was in fact a facet of the original program. It could be said that the decompiler's job is to reproduce the original program's behaviour, including unusual behaviour and bugs.

Locating the handful of program sections that needed modification as shown above was a laborious exercise. Ultimately, two debuggers were run simultaneously, placing binary breakpoints in one, and regular source code breakpoints in the other. The values of key arrays were dumped using memory windows in the debugger, comparing them by eye. Even choosing the arrays to compare was a trial and error process. Perhaps some day a tool could assist with this process, keeping a map of original program variables, and their recompiled equivalents.

5. Results

The deliverable result of this project was the source code recovered through decompilation, and the clients were quite satisfied. Their main concern was to acquire source code for the core algorithms in the mathematically oriented DLL. They received maintainable code that produced the same results, to the last decimal place.

5.1. Recovered Code

It was found that most of the differences between the prototype program (which worked unreliably) and the program to be decompiled (which worked reliably) were quite minor in nature. In several places, C style `malloc` calls were replaced with C++ style `new` calls. Several globals were removed, necessitating a few extra parameters in several procedures.

However, a few significant differences were found, but not where the clients had expected them to be. The clients were quite surprised when an outline was produced of the algorithm that showed the essential difference in a score calculating routine.

Portions of the GUI code were also recovered for the clients, limited mainly by a lack of time that could provided by the authors of this paper. Recall that this was a secondary goal, to be fulfilled only in the unexpected event of being able to fulfil the main goal.

Approximately 1500 lines of code (some 40 procedures) were decompiled in 415 person-hours, not including the first week of exploration. This time included significant software development of Boomerang itself.

In the 135KB math intensive DLL, only about 7KB (5%) was decompilable code; the rest was library code, data, etc. About 50% of this 7KB was decompiled, representing all the code of interest to the clients.

Of the 670KB main executable, 316KB was decompilable code, although this figure includes some code automatically generated by a C++ compiler (e.g. default constructors). Of this, only about 24KB (8%) was decompiled.

Using techniques mentioned in Section 6.1, 78 classes were found in the main executable, compared with 11 in the prototype. 18 of these were dialog boxes, which are possibly easier to rewrite than to decompile.

The code that was not decompiled through lack of time displayed the results in various graphs and tables, recorded and played back speech, handled toolbars and timer messages, and so on. This code, while tedious and expensive to rewrite, contained none of the undocumented core algorithm.

5.2. Sample Output

Figure 8 shows some output from Boomerang before any editing was applied. This example illustrates a variety of things that went wrong.

The parameters are all named and typed; the names and types came directly from the symbol file. Nearly correct source code helped here, but was probably not essential, since a program fragment can usually be understood given sufficient time. Line 4 shows where 8 bytes of memory are allocated on the stack, to be passed as the second parameter to `MoveTo` (lines 7 and 9). The definition and use of `local2` was removed by hand.

Lines 10 and 11 show vestiges of the semantics of push and call instructions. A refinement of Boomerang's dataflow should remove these automatically.

The assignment in lines 14-17 requires some explanation. These lines result from a variation of the idiomatic code shown in Figure 4. `local23 * -2004318071 >> 32` can be replaced by `local23/30`. The second half can be ignored (it is the correction for truncation towards zero; the whole second half is divided by 2^{31}). Making these changes and removing the comments and casts produces `local24 = local23 / 30 + local23 >> 4`, which is correct, and fairly readable.

The right hand side of this large assignment is repeated in the `if` statement of lines 18-21. This is because Booomerang applies as many forward substitutions as possible. Usually, this is a good thing; it tends to build complex expressions from the semantics of individual instruc-

```
 1 void PlotAxes(CDC* pDC, int ptOrigin_x, int ptOrigin_y, int sizePixelsPerTick_cx,
 2   int sizePixelsPerTick_cy, int horizTicks, int vertTicks, int nDrawTicks,
 3   int maxTickSizeX, int arg_24, int maxTickSizeY) {
 4     int local2;  // m[r28{0} - 8]
 5     int local12; // vertTicks{312}
 6     int local26; // r25
 7     CDC_MoveTo(pDC, &local2, ptOrigin_x, ptOrigin_y);
 8     CDC_LineTo(pDC, ptOrigin_x, ptOrigin_y - sizePixelsPerTick_cy * vertTicks);
 9     CDC_MoveTo(pDC, &local2, ptOrigin_x, ptOrigin_y);
10     local11 = local18 - 36;
11     %pc += 6688008;
12     CDC_LineTo(pDC, sizePixelsPerTick_cx * horizTicks + ptOrigin_x, ptOrigin_y);
13     if ((*(char*)(local11 + 68) & 1) != 0) {
14         local26 = (/* opTruncs/u */ (int) (sizePixelsPerTick_cx *
15           -2004318071 >> 32) + sizePixelsPerTick_cx >> 4) + (/* opTruncs/u */
16           (int) (sizePixelsPerTick_cx * -2004318071 >> 32) +
17           sizePixelsPerTick_cx >> 4) / -2147483648;
18         if ((/* opTruncs/u */ (int) (sizePixelsPerTick_cx *
19           -2004318071 >> 32) + sizePixelsPerTick_cx >> 4) + (/* opTruncs/u */
20           (int) (sizePixelsPerTick_cx * -2004318071 >> 32) +
21           sizePixelsPerTick_cx >> 4) / -2147483648 < 2) {
22             local26 = 2;
23         }
24         if (local26 >= maxTickSizeX - (maxTickSizeX < 0 ?  -1 :  0) >> 1) {
25             local26 = (maxTickSizeX - (maxTickSizeX < 0 ?  -1 :  0) >> 1) - 1;
26         }
```

Figure 8: Sample output from Boomerang; unedited, except to fit the page.

tions. In this case, however, it is clearly unhelpful. This problem could be summarised as “when not to propagate”. Some form of common subexpression elimination could solve this problem.

Lines 24 and 25 illustrate another idiom, where a signed integer is divided by a power of 2. A right shift instruction can be used, but if the dividend is negative, it is first incremented. This implements the C requirement that divide operations truncate towards zero.

Once these changes, plus a few changes for readability were made, the final code of Figure 9 was obtained. The reader may come to the conclusion that the decompiler is not contributing much to the overall process; most of the readability is coming from hand editing. While this is true to a degree, it should be remembered that most required source code changes are of the search and replace kind; the original program’s semantics are preserved. Put another way, while readability is (currently) mostly from hand editing, the correctness comes from the decompiler. This was confirmed later in the project, when a few procedures were hand decompiled, and many small errors were introduced.

5.3. Comparison with REC

After the project was completed, a comparison was made with output from the Reverse Engineering Compiler (REC) [2]. The same procedure as shown in Figure 8 was used;

```
void PlotAxes(CDC* pDC,
  POINT ptOrigin, SIZE sizePixelsPerTick,
  int horizTicks, int vertTicks,
  int nDrawTicks, int maxTickSizeX,
  int arg_24, int maxTickSizeY) {
    int nHeight =
      sizePixelsPerTick.cy * nVertTicks;
    int nWidth =
      sizePixelsPerTick.cx * nHorzTicks;
    pDC->MoveTo(ptOrigin);
    pDC->LineTo(ptOrigin.x,
      ptOrigin.y - nHeight);
    pDC->MoveTo(ptOrigin);
    pDC->LineTo(ptOrigin.x + nWidth,
      ptOrigin.y);
    if (nDrawTicks & TICKS_VERT) {
        // Draw Vertical Ticks
        int nTickSize =
          sizePixelsPerTick.cx / 30 +
          sizePixelsPerTick.cx / 16;
        if (nTickSize < 2)
            nTickSize = 2;
        if (nTickSize >= maxTickSizeX/2)
            nTickSize = maxTickSizeX/2-1;
```

Figure 9: Final output for the same code as Figure 8. The code has been edited slightly to fit in one column.

```
 1    ecx = sizePixelsPerTick;
 2    CDC_MoveTo(eax, Ac, ebx);
 3    maxTickSizeY = maxTickSizeY * A34;
 4    A14 = ebx - A34;
 5    ecx = sizePixelsPerTick;
 6    eax = CDC_LineTo(Ac, A14);
 7    ecx = sizePixelsPerTick;
 8    eax = CDC_MoveTo( & A14, Ac, ebx);
 9    ecx = A30;
10    ebp = arg_24 * ecx;
11    ecx = sizePixelsPerTick;
12    eax = CDC_LineTo(ecx + Ac, ebx);
13    if(!(A38 & 1)) {
14        eax = -2004318071;
15        asm("imul ebp");
16        A14 = A14 + ebp >> 4;
17        eax = A14 >> 31;
18        A14 = A14 + eax;
19        ecx = A14;
20        if(ecx < 2) {
21            ecx = 2;
22        }
```

Figure 10: Part of the output from REC, unedited.

a command and a symbol file were produced to provide names and types for parameters. Figure 10 shows a part of the output, corresponding to lines 10-22 of Figure 9, and lines 7-23 of Figure 8.

It does not appear to be possible to specify to REC that a function takes register arguments, or that the callee removes the stack arguments. While compound types such as `sizePixelsPerTick` can be specified (of type SIZE), the member selection (e.g. .cx) is missing (e.g. line 5). For members other than the first, a new name (e.g. `A34`) is used (line 3).

REC assumes that the stack pointer does not change within the function, and as a result, all the stack parameters and temporaries are mixed up. The semantics of individual instructions are obvious, and in some cases are left in a disassembly format (e.g. line 15). Other problems related to dataflow were found (not shown).

To be usable on Windows-based C++ programs, REC would have to be enhanced considerably. Unfortunately, the source code is not publicly available, and development seems to have stopped with version 1.6 in September 2000.

5.4. Partial Source Code

The existence of partial source code for this project certainly sped up the process of recovering usable source code. It provided suitable identifier names and types, as well as occasional comments. It could be asked how much progress would have been possible without this benefit. With the partial source code, it was not necessary to spend a lot of time comprehending the software. If a variable seems to be called `nTickSize`, and its use is not obviously contradicting this name, it can be accepted without further effort that this is a suitable name. Without the partial source code, it is necessary to understand the relationship of the ticks to the other lines, realise the proximity of text labels to some of them, that there are regularly spaced vertical and horizontal lines, and so on, before it is comprehended that these lines are in fact ticks, and that the value of this variable sets the size of those ticks. In this context, partial source code speeds up the process very considerably, but is not strictly essential.

Almost everyone is familiar with graphs and ticks. However, not everyone is familiar with every facet of mathematics. Suppose a decompiler user is not familiar with the Fast Fourier Transform (FFT). To this user, there is a lot of mysterious multiplication and adding going on; to a mathematician familiar with the FFT, there is a transformation from the time domain to the frequency domain. All is not lost, however. Either the decompiler users needs to be skilled in the domain of operation of the program they are working on, or they produce source code with names like “mysteriousMultiplyAdd” which a domain expert can change to “FFT”. Once there is source code for a program, it can be transformed by other people or programs, and these people do not have to be skilled at decompilation.

6. Lessons Learned

The main lesson learned is that decompilation, at least under favourable circumstances, can successfully be used to recover source code in a commercial setting. Some lessons have already been mentioned, such as the importance of types in Windows-based programs (Section 3.4).

6.1. RTTI and Message Maps

One of the most pleasant surprises from this research was the wealth of information supplied by the Run-Time Type Identification (RTTI) system. Briefly, it is at times desirable to be able to identify the type of a class at runtime, and it may be convenient to construct a class of a given named type at runtime. The C++ language supports the former notion directly, e.g. `typeid(p)` is defined to represent the string “`MyClass`” if p points (at runtime) to an object of class `MyClass`. Compilers are able to implement RTTI however they like, and older compilers (such as the one used to compile the executable to be decompiled) do not seem to follow any standard.

Every class with virtual functions and/or RTTI has a hidden member which is a pointer to the Virtual Table (VT). While most elements of that table are pointers to virtual functions, one element points to a special object with run time information, or to a function returning such an object. One of the elements of that object’s class will be a pointer to a C string with the name of the class. It happens that the three main groups of classes in the Microsoft Foundation Classes (MFC) (documents, views, and frame windows) are all generated at runtime using RTTI information. This allows some of these classes to be "serialised", i.e. written to disk in a sensible manner. As a result, the original names of these important classes are stored in the executable file. Given a pointer to such a class, it is often not very difficult to find its original class name. This is very valuable information for a decompiler.

The *Windows* operating system is "message driven". Everything from the movement of the mouse to the destroying of a window is accomplished by sending messages. In order to route messages sensibly to the various objects that can receive them, the MSVC compiler generates structures called message maps. In the object oriented vein, a message is offered to a derived class first, and if it does not process the message, the message is offered to the parent class (in the class hierarchy) next. As a result, the message maps contain pointers from the message map for one class to the message map for that class' parent. Each `CObject` derived class has a virtual function to get the message map for the object. The VT therefore connects the original source code name for a class with that of its parent. As a result, almost the entire class hierarchy is available from any program compiled with the MSVC compiler. It is likely that a similar situation exists for other Windows-based C++ compilers as well.

It seems to be a general principle that the more dynamic features supported by an executable file format, the more high level information there needs to be contained in that executable file format, and as a result, the easier it is to decompile the program to a form close to the original source code. Consider Java bytecode files and .NET assemblies; there have long been decompilers for these formats that can usually decompile such programs to source code that is startlingly similar to the original source code.

6.2. `MoveTo` and `LineTo`

Two common library functions for drawing lines are declared as follows:

```
CPoint MoveTo( int x, int y );
BOOL   LineTo( int x, int y );
```

Each takes an `x` and `y` logical coordinate; the return values are normally ignored. It came as a surprise therefore when it was found that calls to `LineTo` were preceded by the expected two push instructions (the implicit `this` parameter is passed in a register), but `MoveTo` was always preceded by three push instructions.

The mystery is resolved by considering the return value of `MoveTo`. `CPoint` is a structure, returned by value. The way that compilers return structures is part of the calling convention. The MSVC compiler passes the address of the returned structure as a (second) hidden parameter. Even though almost every call to `MoveTo` will ignore the return address, the compiler has to allocate 8 bytes for the return value, and push the address of this memory every time the function is called. The return value for `LineTo` is in a register; there is no cost for ignoring its value.

When `MoveTo` was designed (presumably back in version 1.0 of *Windows*), the designers probably did not consider the cost of returning a value (not by reference) that is so rarely used; it contains the position last moved or drawn to. Backwards compatibility requires that this cost has to be borne forever more.

7. Future work

Certainly, this project highlighted several aspects of Boomerang that needed improving. The less mathematical parts of the code were very dependent on types. While Boomerang's handling of types is much improved as a result of this project, type inferencing is planned for the near future. Type inferencing is the determination of the types of variables from the semantics of instructions, combined with sources of type information such as calls to library functions. This will likely have a huge impact on the quality of decompiled programs.

Several problems were due to a lack of maturity in Boomerang. Examples include the remnants of push and call instructions. These require no new theory, merely time and attention to detail.

The surprising ability to recover class names and the class hierarchy makes it important that some kind of compiler agnostic way be found for taking advantage of Run Time Type Identification, message maps, and the like. Such information will likely lead to more entry points, which could allow greater automation of the decompilation process.

Section 3.2 highlighted the need for the ability to identify statically linked library functions. This will remove the present need for much manual typing of symbolic information.

The C++ language is now well enough established that legacy applications typically are written in that language. There is a consequent need to be able to represent class information properly. Other features of C++ that require some consideration include exception handling, and the ability to remove code automatically inserted by the compiler (e.g. constructing and deleting certain temporary objects, destructing local objects that go out of scope, etc.)

A small number of "thunks" were found during this project. Thunks are very short pieces of code, often only one instruction plus a jump instruction, that are needed for certain housekeeping functions. For example, an offset commonly has to be added to the `this` pointer, to cast it from a pointer to one kind of object, into a pointer to another type of object. It seems likely be that dealing with such thunks correctly will become important.

8. Conclusions

Critical source code for the client's Windows-based application was recovered. In addition, source code for several noncritical but desirable procedures was also recovered. The recovery used a combination of the commercial disassembler IDA Pro, the Boomerang decompiler, and considerable hand editing. The recovered source code was readable, maintainable, and directly suitable for use by the client in their rewriting of their application. The recovery would have been much more difficult without the availability of partial source code.

Boomerang, while found to be in need of many improvements, provided the foundation for the decompilation process. Valuable support for the names and types of structure members was added during the project, paving the way for full type inferencing in the near future. By the end of the project, the authors could decompile most code by hand, with the possible exception of the trickier branch statements. Despite the considerable editing that was needed to the decompiled output, using Boomerang was still much safer than hand decompilation, because of the high probability of errors with a completely manual approach.

It is believed that this project demonstrates that decompilation is capable of solving future business problems. During the Y2K maintenance process, it was estimated that up to 5% of all source code is missing. In most cases, the only alternative (rewriting from scratch) is very expensive, so that even moderately labour intensive solutions would be economical.

Boomerang should soon have full type inferencing, and analyses for transforming virtual function calls into class member function calls. Perhaps this project, together with such advances, will show that decompilation is not like "making pigs from sausages" [18].

References

[1] Boomerang web page. BSD licensed software, 2002. `http://boomerang.sourceforge.net`.

[2] G. Caprino. REC - Reverse Engineering Compiler. Binaries free for any use, 1998. Retrieved Jan 2003 from `http://www.backerstreet.com/rec/rec.htm`.

[3] C. Cifuentes. *Reverse Compilation Techniques*. PhD dissertation, Queensland University of Technology, School of Computing Science, July 1994. `http://www.itee.uq.edu.au/~cristina/dcc/decompilation_thesis.ps.gz`.

[4] C. Cifuentes. The dcc decompiler. GPL licensed software, 1996. Retrieved Mar 2002 from `http://www.itee.uq.edu.au/~cristina/dcc.html`.

[5] C. Cifuentes and K.J. Gough. Decompilation of binary programs. *Software - Practice and Experience*, 25(7):811–829, 1995.

[6] DataRescue. IDA Pro, 1998. Retrieved Jan 2003 from `http://www.datarescue.com/idabase`.

[7] DeCompilation wiki page, 2001. `http://www.program-transformation.org/Transform/DeCompilation`.

[8] T. Hoffman. Recovery firm hot on heels of missing source code. In *Computer World*, 24th March 1997.

[9] *Pentium Processor Family Developer's Manual*, volume 3, chapter 25, pages 25-145 – 25-147. Intel Literature, 1995.

[10] Intel Architecture Software Development Manual, 1997. Pages 3-182 - 3-185, retrieved July 2004 from `www.intel.com/design/pentium/manuals/24319101.pdf`.

[11] A. Mycroft. Type-based decompilation. In S. Swierstra, editor, *8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, Amsterdam, Netherlands, March 1999. Springer-Verlag.

[12] J. O'Gorman. *Systematic Decompilation*. PhD thesis, University of Limerick, 1991. Technical Report UL-CSIS-91-12 Retrieved Mar 2002 from `ftp://www.csis.ul.ie/techrpts/ul-91-12.ps`.

[13] Source Recovery's HP-UX C/C++ Decompiler, 2002. Retrieved Feb 2004 from `http://www.sourcerecovery.com/abstract.htm`.

[14] The Source Recovery Company, 1996. Retrieved July 2004 from `http://www.source-recovery.com`.

[15] M. Van Emmerik. Identifying library functions in executable files using patterns. In *Proc. Australian Software Engineering Conference*, pages 90–97, Adelaide, Australia, Nov 1998. IEEE-CS Press.

[16] L. Vinciguerra, L. Wills, N. Kejriwal, P. Martino, and R. Vinciguerra. An experimentation framework for evaluating disassembly and decompilation tools for C++ and Java. In *Proc. Working Conference on Reverse Engineering*, pages 14–23, Victoria, Canada, Nov 2003. IEEE-CS Press.

[17] M. Ward. Assembler to C migration using the FermaT transformation system. In *Proc. International Conference on Software Maintenance*, pages 67–76, Oxford, England, 1999.

[18] M. Ward. Pigs from sausages? Reengineering from assembler to C via FermaT transformations. *Science of Computer Prgramming Special Issue: Transformations Everywhere*, 52(1-3):213–255, 2004.