# A brief note on the reverse engineering of protected computer programs from a UK perspective

Matthew Gream <matthew.gream@pobox.com>
February 2003



## Scope and purpose

This is a brief note about legal and technical perspectives of reverse engineering of protected computer programs under UK law.

It is intended to be brief and sweeping, yet provide various references for further investigation.

## Legal perspective

The right to reverse engineer a protected computer program is available by way of the following legislation:

- **International law:** Article 9 of the WTO TRIPS Agreement, which specifies that copyright protection shall extend to expressions and not ideas. Article 2 and Article 4 of the WIPO Copyright Treaty, which specifies that copyright shall extend to expressions and not ideas.
- **Community law:** Article 5 of the Council Directive 91/250/EEC of 14 May 1991 on the legal protection of computer programs (as amended), which allows a person to "observe, study or test the functioning of the program in order to determine the ideas and principles which underlie any element of the program". Article 6 of the Council Directive 91/250/EEC of 14 May 1991 on the legal protection of computer programs (as amended), which allows the specific act of Decompilation for the purpose of creating interoperability.
- **National law:** Section 296A of the Copyright, Designs and Patents Act 1988 (as amended), which implements Article 5 of EEC Directive 91/250. Section 50B of the Copyright, Designs and Patents Act 1988 (as amended), which implements Article 6 of EEC Directive 91/250.

There are three significant reasons for allowing a right to reverse engineer, and these are:

- **For interoperability:** to allow fair competition and prevent "derogation from grant" (e.g. competitive products, "spare parts and repair"). This is only available where there are no existing provisions for interoperability (i.e. the owner of the work has not published interfacing details). This is provided for in EU/UK legislation (EEC Dir 91/250 Art. 6 & UK CDPA s50B) and case law, but under very specific conditions.
- **To preserve the idea/expression dichotomy:** by ensuring that copyright in the computer program does not restrict any other property rights that may exist (e.g. patented algorithms, utility models, embedded works by third parties). This is implicitly required by signatories to the TRIPS agreement, and codified in EU/UK legislation (EEC Dir 91/250 Art. 5 & UK CDPA s296A).
- **To achieve performance of objective:** when decompilation is a necessary part of utilising the work according to the express/implied terms of the license. This would be rare, as few computer programs would require decompilation as a necessary part of use. This is an established legal principle in many legal systems.

For a very good legal overview of reverse engineering, which includes international aspects, refer to REVERSE ENGINEERING & DECOMPILATION OF COMPUTER PROGRAMS (The legitimate boundaries of Copyright Protection) by Aziz ur Rehman, Hafiz.

Other resources include Reverse Engineering Clauses in Current Shrinkwrap and Clickwrap Contracts, THE LAW & ECONOMICS OF REVERSE ENGINEERING, and REVERSE ENGINEERING UNDER SIEGE.

## Technical perspective

There are numerous resources focused on reverse engineering of computer programs from a technical perspective, such as:

- Tools for decompilation (e.g. dcc, Mocha, Anakrino, Perl).
- Commercial services (e.g. MicroAPL, Springstone Software).
- Publications on decompilation techniques (e.g. Win32, Unix).
- Academic conferences and workshops (e.g. WCRE2001).
- Collections of resources (e.g. Softpanorama, Reverse Engineering, Program-Transformation.Org, Open Directory, searchVB).
- Discussion forums and communities (e.g. Decompiler.com).
- Protection against decompilation (e.g. Dotfuscator, Robust Obfuscation).

In terms of the various ways that a computer program is amenable to reverse engineering, these are a few salient points:

1. The binary components that form part of a computer program are often in structured formats. An executable file format (e.g. Executable File Formats) often has separate code and non-code segments, and there can be multiple segments, which can aid reverse engineering by indicating higher-level boundaries and arrangements.
2. The executable (or non-executable) components may contain debugging information, such as program symbols (e.g. Release mode debugging with VC++). *Typically* these are used only for internal development, and not for production releases. But if they are present, then they can provide significant information about the structure of the source code and its relationship to the object code.
3. The compiler that converts source code to object code is deterministic and structured, so making reverse inferences is not difficult, however production releases are typically "optimised" which does cause obfuscation. Source code is very structured, and that structure translates strongly to object code, however any particular translation is a function of the particular compiler, its settings, and its environment (while remaining deterministic).
4. Single, or related, modules or libraries, are typically linked into a final executable image, and as software is typically "modularised", then all of the functionality for a particular purpose (e.g. a set of modules for an interoperable interface) are often located within single contiguous area of the image. This means that analysis can be scoped to a small part of an otherwise large computer program.
5. During execution of a computer program, the execution path will "jump" to other routines within the image as it reuses code from other parts of the program: it is possible to locate these transfers of control and detect the way that arguments are passed, and this may help extract "interface functions".
6. Computer programs typically rely upon operating system primitives, or platform libraries, such as for file services or string and time processing. These are highly distinguishable transfers of control in execution and can be detected using third party tools (e.g. Unix truss), utilities supplied by the operating system, or the vendor's development suites.
7. Computer programs are sometimes released as one single executable image, or a number of libraries that collaborate together: these libraries often export well-defined modular interfaces, which include symbolic naming information (e.g. DLL symbols).
8. Interoperable protocols (such as RPC, CORBA and XML-RPC) are often implemented using third-party libraries (e.g. omniORB) (making it possible to analyse the way a computer program uses those libraries as mentioned above) and/or involve the generation of "custom code" (which is typically highly structured, and results in predictable and understandable object code more amenable to reverse engineering).
9. Interoperable protocols are often standardised and involve data communication protocols that can be observed and used as part of the

reverse engineering process. There are numerous protocol analysers available for the purpose, including “open source” (e.g. Ethereal).

10. Decompilation and reverse engineering occurs on a regular basis in the community, especially in the field of computer security (e.g. BugTraq).
11. Some languages (e.g. Java, Python or Perl) compile from source code to an intermediate format or ‘bytecode’ that has greater structure and less complexity than native platform assembly language. This can make decompilation easier (e.g. perl2exe). This is a very significant concern for Java (e.g. Advice on protection, Obfuscation techniques).