Protecting Source Code with ANDF

Stavros Macrakis
OSF Research Institute

January 25, 1993

Binary protects source code but is architecture-dependent. ANDF protects source equally well, but is architecture-independent.

1. Introduction

Software source code is a highly valuable commodity, and software vendors go to great lengths to protect it. In particular, software is generally distributed in only object form, which is copyable and executable but does not give away engineering methods and techniques. Binary is, however, specific to each target architecture. OSF's Architecture-Neutral Distribution Format (ANDF) is independent of target architecture yet protects engineering methods and techniques.[1]

Software is typically written in a high-level programming language, which makes it easy to design, create, understand, modify, and correct the program; translating (compiling) it to object form makes it efficient to execute. Compilation removes most of the information needed to understand the program, and therefore to modify it, reuse its components, or take its best implementation ideas.

1. ANDF is based on the TDF technology from the U.K. Defense Research Agency.

The ANDF technology introduces a third form of a program: the architecture-neutral ANDF form. ANDF contains just the information needed to compile a program on a variety of machines, and therefore excludes the information needed to modify it or reuse its components.

This paper discusses the information carried by ANDF form, and possible reverse engineering attacks on programs starting from binary and from ANDF. ANDF provides somewhat more information about the source program than does binary, but in none of the attack scenarios does this make a substantive difference in the ease of attack.

2. The value of source code

Software source code is complex, expensive, and valuable.

Software is a complex engineered product: source code embodies much of the detailed engineering design, and binary code is the finished product. As with other engineered products, the design documents are valuable intellectual property, and software vendors guard them closely as trade secrets.

Continuing engineering on a product requires access to the design documentation. With the design documentation, functionality can be enhanced, design elements can be reused for other products, and new interfaces can be added.

Source code is expensive: it reflects the extensive engineering work necessary to take an idea from a concept to a working product; it contains ameliorations invented to improve performance based on analysis and experience; it incorporates corrections to errors discovered through testing and actual usage.

Source code is valuable: with it, variants can be created for particular market requirements; with it, new products can build incrementally on old ones; with it, engineering techniques used in one product can be applied to others; with it, interfaces can be created between products. Without it, programs cannot evolve to meet new needs.

Losing control of source code can be costly. With knowledge of source code, a competitor can incorporate much of the know-how embedded in an application into his own; he can study competitors' weaknesses and exploit them in his own products; he can create interfaces which allow his products to piggy-back on the other.

Thus, application developers protect their source code jealously.

Protection of source code

Source code is protected by limiting its distribution to those with a "need to know", by using legal protection mechanisms (law of copyright and trade secret), and by minimizing the information about it in object code. (In particular, object code is invariably stripped of its symbol table.)

3. Overview of software artifacts

So far, we have been talking of program source and object. Actually, software that is designed to be maintained over time usually has many other pieces, starting with design documentation.

Design documentation takes many forms. Some software producers have very formal procedures, where every function and every module has an associated design document defining its interfaces and behavior. Other software producers rely on more informal design documentation. Comments in the code itself often replace separate design documentation.

Software has build files (Makefiles in Unix) which describe how the system is put together. Some source components are generated automatically from specifications written in some other language; for instance, most compilers' parsers are generated by parser generator programs directly from the specification of the grammar.

Revision control (SCCS, RCS, etc.) keeps track of program versions. As programs evolve, revision control information is important to keeping track of bugs, bug fixes, and changes.

Libraries of tests ensure that old bugs have been fixed, and catch some new bugs.

Finally, we come to source code itself. Source code must compile into running object code. At the same time, it must be understandable by the programmer writing the code and those maintaining it. Thus it contains many hints about the programmer's intent, some explicit--variable and function names, comments, and type definitions--and others implicit, like the choice of looping constructs, and the scope of variables.

To be executed, source code must be translated into object code, which is machine- and environment- dependent.[2] Object code contains only that information needed by the computer to execute the program, and not the information needed to understand it or modify it.

4. Reverse engineering and decompiling

In any branch of engineering, *reverse engineering* takes a finished product and recreates the engineering knowledge necessary to reproduce it. In the case of software, simply reproducing is trivial (you just copy the bits)[3]; what is difficult is recreating the engineering knowledge necessary to modify, cannibalize, or interface to a program. So a good definition is:

> Reverse engineering of software means reconstructing sufficient engineering knowledge to modify a program, reuse parts of it in other contexts, and interface to it.

Looking at the software process as a whole, reconstructing source is only part of reverse engineering, since source does not in general contain enough engineering knowledge. In the case of a spreadsheet, for instance, the formula interpreter may have been generated automatically (by a tool like yacc) from a grammar description; it may be very difficult to add a new operator by hand to the automatically generated state machine.

Indeed, reconstructing engineering knowledge may even be necessary when source code *is* available. In software engineering, "reverse engineering" in

2. I use "object code" to refer to programs that are ready to load (a.out, not .o).
3. This is a significant difference from manufactured products, where recreating manufacturing processes and tools is non-trivial.

fact typically refers to the process of understanding *source code* better, not reconstructing source code from object code!

Still, reconstructing source from object is an important step in reverse engineering of object code, and is usually called *decompiling*. There has been some research on decompiling, and the technology is fairly mature: from an arbitrary object binary, a decompiler can reconstruct a program in a source language which has the same effect.[4] This typically is done by reconstructing straight-line code, and then analyzing control flow to abstract traditional control structures. Although the result is more or less readable as code, its intent is usually difficult to understand in the absence of meaningful variable names, comments, and so on.

5. Reverse engineering of ANDF

Now, the issue is whether ANDF provides additional information which makes reverse engineering easier or more effective.

The key observation is that no source or intermediate form of a program can be *less* easy to reverse engineer than its object binary: if it were, it would suffice to complete the installation of binary, and then reverse engineer the binary.

Thus the question becomes: what *additional* information does ANDF provide that object binary does not?

The next section will show that decompiling ANDF into C is somewhat easier than translating from binary to C (especially for data structures), and produces a bit more information on control and data structures. On the other hand, the resulting C program is close to incomprehensible for any realistic commercial program, just as is a C program decompiled from binary.

Structure and presentation of ANDF

For an introduction to the internal structure of ANDF, please refer to the paper *The Structure of ANDF: Principles and Examples*[5]. That paper goes

4. Products include “decomp” for the VAX, “exe-to-C” for MS-DOS.

into detail on the mechanisms used by ANDF to represent various language constructs.

For this paper, the detailed structure of ANDF is not relevant, and we will use examples in pseudo-C syntax which reflect the information carried in the ANDF without the details. Additional information contained in ANDF itself guarantees C semantics (or Fortran or Ada or Cobol semantics, as the case may be) for various constructs, but does not contain additional information about the source code.

6. Decompiling ANDF to C

ANDF can be decompiled to C just as object code can. In both cases, this requires intimate knowledge of the language to be decompiled. For machine language, the instruction set architecture must be well understood, and tools for interpreting instruction formats must be available (disassemblers). For ANDF, the ANDF language must be well understood, and tools for decoding it must be available (ANDF readers).

Although ANDF readers are more difficult to construct than disassemblers, and certainly not widely available, we will assume that a decompiler has access to this tool and to the complete specification. Without the tools and specification, reliable decompiling would be almost impossible, especially since the encoded form of ANDF is tightly packed in an unfamiliar way.

Given such tools, we examine how much easier it is to decompile ANDF than object code.

Module and function structure

C programs have a simple module structure: top-level function and object declarations can be either local to a source file or global. The separation into files gives some hint as to the global structure of the program. ANDF has essentially the same module structure. Object code has no module structure.

5. *The Structure of ANDF: Principles and Examples* is an introduction to the structure of the ANDF interface through graduated examples and is written for language implementors and designers. It is available from OSF as report number RI-ANDF-TP1-1.

As a general rule, functions which are contiguous in source modules are contiguous in ANDF and object modules. This gives a small amount of information about the source code's structure.

ANDF does preserve information about the beginnings and ends of functions, unlike object code. However, on most architectures, procedure entry and exit code is rather stereotyped and can be identified automatically even in object code.

ANDF also preserves explicit references to standard libraries. Decoding this does require understanding token definition libraries, but once this is done, both macro and procedural references to libraries can be resolved easily. Procedural references to standard libraries in object code are relatively easy to decompile, since the same libraries are usually available separately with their named entry points; shared libraries make this even easier. On the other hand, macro entries are more opaque.

Thus, ANDF does give additional information about the module structure of an application. In principle, tools could be written to hide this structure, but they do not exist today. ANDF also gives additional information about standard library references.

Straight-line executable code

Straight-line executable code is notably easy to reconstruct from object code, as from ANDF. Straight-line code reconstructed from ANDF is likely to be closer to source because fewer optimizations are performed in ANDF producers than in optimizing compilers.

Control structures

Programs written in higher-level languages use a variety of control constructs in order to express the programmer's intention clearly. Thus, in C, loops can be expressed as while loops, do/while loops, for loops, and of course combinations of goto's. C labels are local to a procedure. ANDF flattens the space, and provides only goto's and unconditional repeat loops (of the form `while (1) {...}`).

In most architectures, object code uses only (conditional) goto's, and their scope is global.

Unlike binary, ANDF does identify loops, but not the choice of loop constructs. In binary, all backwards jumps must be analyzed to reconstruct loops.

Representation of elementary types

Source, ANDF, and binary must somehow represent much of the same information about data types. For instance, every arithmetic operator must specify which operation to execute (addition, multiplication, *etc.*), and on what data type. In C, the data type is specified in the data declaration, and the operation in the executable statement, *e.g.*:

```
int net, gross, expenses; ...
net = gross - expenses;
```

In abstract ANDF (as it comes out of the producer), the data type is specified in the data declaration, and the type of operation in the executable statement. The implementation of the data type is specified by the installer, *e.g.*:

```
int v0, v1, v2; ...
v0 = integer_subtract(v1, v2);
```

Abstract (tokenized) ANDF does distinguish between data types which may be implemented differently on different machines (*e.g.* `long` and `integer`), whereas object code only preserves differences relevant to its architecture.

In concrete ANDF -- after target-specific tokens have been instantiated by the installer -- the data size is specified in the data declaration, and the type of operation in the executable statement, *e.g.*:

```
int[-32768..32767] v0, v1, v2; ...
v0 = integer_subtract(v1, v2);
```

In object code, the individual data items are typically not declared separately, and the type and size of the operation is specified in the executable statement:

```
bytes v[6] ...
v[0..1] =
       integer_subtract_16(v[2..3], vv[4..5]);
```

Thus, for scalar variables, essentially the same information is available (except of course the names), although in the ANDF it is in principle easier to separate variables than in binary. In practice, it is rather easy to determine variable slots, and even their type, from binary. Any typedefs and macros used to give mnemonic names to data types are unavailable both in object and in ANDF.

Compound data types

Most interesting data structures are composed of compound data types such as records and unions. In source code, the names of fields and their types are important documentation. For instance, a cell in a sparse spreadsheet might be represented as:

```
struct cell {
     struct cell *up, *down, *right, *left;
     formula contents;
     display_style style;
        }
```

This expresses the intent of the program writer clearly and simply. In ANDF, most of the intent is gone. The ANDF equivalent is roughly (depending on what `formula` and `display_style` are):

```
struct s1 {
   void *f1;
   void *f2;
   void *f3;
   void *f4;
   void *f5;
   void *f6;
    }
```

Actually, the information is not quite this explicit in the ANDF, but it is relatively easy to reconstruct it in the case of simple records. Variant records (unions) are more difficult.

In object code, reconstructing the structure of records is less straightforward.

Thus, ANDF gives considerably more information about the general shape of compound data structures than does object code: reconstructing the concrete representation of data types is relatively straightforward. On the other hand, it gives no information about the *intent* or *meaning* of that shape, as do declarations in the source, which use typedefs, struct names, and so on to document usage.

Summary: decompiling ANDF

ANDF gives a decompiler several additional pieces of information compared to binary object code. The division into procedures and references to standard libraries are clear and unambiguous. Control structures and in-line code are somewhat more transparent, but can generally be reconstructed even from optimized binary code by a good decompiler. Data types are probably the area where ANDF gives the most additional information, but here too the information is low-level.

All this analysis presupposes a mastery of ANDF specification and know-how. Although this mastery does not exist today, we must assume that it will in the future.

7. Reverse engineering from decompiled code

Decompiled object code or ANDF is functionally equivalent to the original code. That is, it contains all information that is useful to the compiler. However, it contains almost none of the information that is useful to a programmer who wishes to understand the program.

Indeed, one proposed way of protecting source code is “shrouding” it, that is, performing a source-to-source transformation that removes human-oriented content. This includes comments, formatting, and identifier names as well as higher-level control structures and macros.

Although formatting and some higher-level control structures can be reconstructed by a decompiler, comments and identifier names cannot. There is a large difference in readability between code with meaningful names and comments and that without. Consider, for instance, the following very simple code fragment from a fictional compiler:

```
case 23:
   { v42 = f71(v3.r5);
     f56(12, v1, v42.r23);
     if (v42.r3 != 0)
        f56(12, v1 + 1, v42.r23 + 1);
     return 0;
    }
```

and compare it to the form a programmer would have written:

```
case float_to_float:
/* Copy floating point number */
   { dest_reg_num = get_float_reg_num(dest.where);
     issue_inst(float_copy, dest_reg_num,
                                reg_num(src_reg));
     if (dest.doublep)
        issue_inst(float_copy, dest_reg_num + 1,
                               reg_num(src_reg) + 1);
     return SUCCESS; }
```

This case might be one of a hundred in a single routine. In the absence of any explanatory commentary or design documentation, the decompiled code is useless.

There do exist tools to help maintenance programmers understand existing, poorly-documented programs. Patient application of these tools to decompiled code (from object or from ANDF) could, with time, yield results. It is doubtful that these results would be very useful.

8. The goals of reverse engineers

Having seen what decompiled code might look like, let us consider some of the putative applications of decompiling and reverse engineering, and see if decompiled ANDF can in fact be useful for them.

This is a purely technical discussion; from a legal point of view, most of these techniques would surely constitute infringements on the intellectual property rights of the original software vendor.

Extracting algorithms

Some companies are known for having particularly efficient algorithms in certain critical areas. A competitor may want to use that same algorithm in a different product. In order to do this, he must understand the algorithm well enough to adapt it to his own data structures and interfaces.

To do this, the competitor would first have to find this algorithm in the middle of the code. This can be done by profiling in some cases; in other cases, large parts of the code would have to be studied.

After the algorithm is located, it will have to be decompiled. This will give unreadable code (see the section on decompilation). Now, reverse engineering proper is needed to understand what this code does.

Although published algorithms are usually short and clear, practical implementations are often long and obscure. For instance, the inner loop of the original Boyer-Moore fast string search algorithm consists of two lines of pseudo-Algol. The implementation in GNU Emacs is about 125 lines of

C code in which Emacs's data structures and the Boyer-Moore algorithm are woven together. Understanding the algorithm requires understanding the usage of the data structures. The C code itself is difficult to understand, and thus is heavily commented. Comments would of course be unavailable to anyone reverse engineering the algorithm. Moreover, any particular implementation of an algorithm may depend on invariants of data structures or coding conventions which are not evident from examination of the decompiled code.

It is clear that deciphering such code is a non-trivial exercise. On the other hand, if the algorithm's implementation is short and simple, it is likely that it can be just as well decompiled and reverse engineered from object code.

Extracting data formats and interfaces

Another reason to reverse-engineer a piece of software is to understand its external interfaces. A vendor may use a proprietary data format to exclude competitors from producing plug-compatible software.

Some data formats are readily reverse engineered, although the process is laborious. For a word processing program, a paragraph will typically be represented with some sort of preamble containing style parameters such as typeface (with values drawn from the font list), size (with small integer values), lightness (with values drawn from light/normal/bold), etc. To reverse engineer this format, one creates paragraphs with a variety of formats, varying one parameter at a time, and sees how they are represented in binary.

Other data formats (especially those based on the internal representation of the object) are more difficult. They may contain pointers to other parts of the structure, which only make sense if the memory organization of the application is understood. Or they may contain additional internal data which must be consistent with user-visible data, but is not directly visible to the user. For instance, there may be a special code for a given font in a given size and lightness.

In these cases, understanding the underlying program is usually necessary. Most of the information is contained in the program's data declarations and the attached comments. In the absence of meaningful record field names and

comments, it is usually unenlightening to know that a particular data structure consists of some large number of short integer fields, integer fields, and long integer fields.

Still, given sufficient economic motivation, it is perfectly possible to reverse engineer such interfaces from binary or from ANDF.[6]

Cannibalizing code

Reverse engineering might be useful as a way of avoiding developing code from scratch. In fact, it is often possible to lift object code directly and reuse it. But this will only work if the interface is well-understood. For instance, one could feel confident that a `sin` routine for a given architecture would work if lifted bodily into another application on the same architecture. The same is true in ANDF.

On the other hand, it is highly unlikely that any major module of a large program could be reused in this way, or through decompilation. The main problem is that the interface is not fully specified: the precise assumptions being made, the data structures being used, and the conventions being respected.

Competitive analysis

Reverse engineering could be used to find weak points in a competitor's products, or latent but unpublicized functionality.

Finding weak points by reading fully documented code is difficult; trying to find them by reading decompiled code is unrealistic. System testing is probably a far more effective way of finding weak points.

Latent functionality *is* an area where reverse engineering may be useful. Once a critical routine is identified, it may well be possible to notice a path that is only taken under special circumstances. This can also be done using a debugger, but decompiled code is probably an easier way.

6. For example, Accolade Inc. reverse engineered Sega's game interface using decompilation. See Electronic News **38**:1928:18 (Sept. 7, 1992).

Of course, any latent functionality which is conditionally compiled will be completely invisible to both ANDF and binary versions.

Writing clones

Finally, a decompiled program might be used to write a clone of the original program. That is, the whole of the decompiled program would be used as a basis for a new version.

This is highly unrealistic. Maintaining fully-documented source code is difficult enough. Maintaining poorly-documented source code is a well-known problem. Maintaining decompiled source code is unthinkable.

9. Conclusion

Like binary object code, ANDF provides sufficient information to execute a program. Unlike binary, it provides it in an architecture-neutral form, allowing installation on a variety of machine types.

Like binary, ANDF can be decompiled into an equivalent C program. In fact, in some areas it makes decompilation somewhat easier:

- ANDF executable code is more structured than object code. However, object code decompilation is well-enough understood that ANDF provides no additional useful information.
- In order to be installable on different machines, the ANDF form carries machine-independent data definitions rather than committing to specific layouts. These data definitions mirror the overall structure of source code data definitions more closely than does object code.
- ANDF library calls are more explicit than object code library calls. But anyone with access to standard object libraries can extract the same information from object code.

But reverse engineering is much more than just decompilation. Decompilation produces a program with meaningless identifiers, no comments, and no higher-level data definitions. Makefiles, test cases, and design documents are all missing.

Useful reverse engineering requires understanding the functioning of the program in order to re-engineer it. ANDF provides no more help here than does binary.

Thus, ANDF allows distributing architecture-neutral code with no compromise to source integrity.

For further information please contact:

Stavros Macrakis
macrakis@osf.org
(617) 621-7356

Copyright 1993 by Open Software Foundation, Inc.

All Rights Reserved

Permission to reproduce this document without fee is hereby granted, provided that the copyright notice and this permission notice appear in all copies or derivative works. OSF MAKES NO WARRRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. OSF shall not be liable for errors contained herein or for any direct or indirect, incidental, special or consequential damages in connection with the furnishing, performance, or use of this material.