University of Dayton Law Review

Volume 19 | Number 3 Article 3

4-1-1994

Software Reverse Engineering in the Real World

Andrew Johnson-Laird

SOFTWARE REVERSE ENGINEERING IN THE REAL WORLD

*Andrew Johnson-Laird**

I. INTRODUCTION

One legal definition of reverse engineering is "a fair and honest means of starting with the known product and working backwards to divine the process which aided in its development or manufacture."[1] In *Secure Services Technology, Inc. v. Time & Space Processing*,[2] a United States District Court defined reverse engineering as "the process of starting with a finished product and working backwards to analyze how the product operates or it was made."[3] Both definitions focus on the process and not the product of reverse engineering. This point is overlooked by those who seek to ban software reverse engineering because it has the capability of being used (with difficulty) to produce infringing copies of original programs. In the mechanical world, reverse engineering, taking something apart to see what makes it tick, is a well established principle. In fact, it is considered prudent to reverse engineer a competitive product specifically to avoid infringing any patented technology the product may contain.

Software reverse engineering, also called "decompilation,"[4] differs from mechanical reverse engineering. First, intermediary copies of the original software must be made and second, to a much larger degree, it is an *additive* process. The programmer starts with the lowest possible level of abstraction devoid of any higher level information, and then adds personal knowledge and experience. Software reverse engineering is difficult and time consuming. It represents a remedy of last resort for obtaining information not otherwise available. It is not the preferred

---

* President, Johnson-Laird, Inc., Portland, Oregon. Johnson-Laird, Inc. specializes in the preservation and analysis of computer-based evidence, including digitized audiovisual data, for such purposes as plagiarism assessment, misappropriation of trade secrets and patent infringment.

1. Kewanee Oil Co. v. Bicron Corp., 416 U.S. 470, 476 (1974). Although this case addressed the conflict between state and federal laws regarding trade secrecy, this definition is valid when applied to software reverse engineering.

2. 722 F. Supp. 1354, 1361 n.16 (E.D. Va. 1989). This was a computer software copyright infringement case, hence, perhaps, the definition is more appropriate to the process of software reverse engineering.

3. *Id.*

4. Decompilation is, in fact, a mythical process. It appears to have sprung into use as an antonym for "compilation" – the process of transforming the human readable form of computer software into the form that can actually be run on a computer. While the creation of the word to describe the reverse process is easy, the process of decompilation is impossible.

method for "fast buck" software thieves. Thieves have no need of reverse engineering; their objective can more easily be accomplished by outright copying of the original diskettes and manuals.

Opponents of software reverse engineering, perhaps playing on the fact that many in the legal profession and the judiciary *think* that they may not understand software reverse engineering, have been indulging in technological fear-mongering. They have presented arguments against reverse engineering that, on closer technical examination, can be seen to be sophistry superimposed on technical ignorance of software development. Recent developments in Japan, where new copyright law is being considered which explicitly permits reverse engineering, have prompted the appearance of disinformation, even in respected newspapers:

> Reverse engineering essentially allows a company to take a software program, break it down into the ones and zeroes of computer language and then essentially duplicate it.[5]

> United States companies, like IBM, say they are particularly concerned about a form of reverse engineering known as "decompilation." In that procedure, software engineers "translate" a computer program's ones and zeroes of binary code into a more readable language. That translated version can be easily modified and "recompiled" into a new program that is only slightly different from the original, a prospect that unnerves many U.S. software companies.[6]

> American officials say decompilation involves copying software, which is illegal. There are legal ways, they say, to find out how a program works. In addition, they say, once a program has been decompiled it can be changed somewhat and recompiled into a new program in a way that makes it hard to tell whether the original had been copied.[7]

These quotations reflect a jaw-dropping degree of ignorance of the process of reverse engineering and ascribe magical and mythical qualities to "decompilation." As the old adage goes, one cannot believe what one reads in the newspapers; this should be extended to journalists who no longer should believe what they read in press releases.

This Article will attempt to show that, although the *process* of software reverse engineering is difficult, it is not difficult to understand what software reverse engineering is. Additionally, this article will ex-

---

5. David P. Hamilton, *U.S. Criticizes Japan On Panel Software*, WALL ST. J., Nov. 10, 1993, at B5.

6. *Id.*

7. Andrew Pollack, *U.S. Protesting Japan's Plan to Revise Software Protection*, N.Y. TIMES, Nov. 22, 1993, at D2.

plain why every computer programmer uses this process at one time or another. Finally, this article will explain what information can and cannot be obtained by software reverse engineering. It is beyond the remit of this article to address the issue of whether or not software reverse engineering is legal under current law and what level of similarity may exist without infringement. This paper will explain software reverse engineering by using common everyday paradigms. Additionally, this paper will provide a detailed example of software reverse engineering, with sufficient intellectual life support to permit non-programmers, including journalists, to involve themselves in the experience. Finally, this paper addresses some of the technically erroneous beliefs propagated by those who oppose the technologically necessary process of reverse engineering.

II. The Issues

A. The Right To Create Competitive Or Compatible Products

For a software developer, the crux of the software reverse engineering issue is expressible in two questions. First, is it appropriate and/or legal for a developer to create software that competes directly in the marketplace with an existing successful program? Second, is it appropriate and/or legal for a developer to create software that works with the data used by another existing successful program, effectively augmenting the capabilities of the existing program but with the self-serving effect of being attractive to an existing body of users of the existing program?

If the answers to these questions are yes, then software reverse engineering is a *necessary* process. Accordingly, reverse engineering and the production of intermediary copies under the "fair use" provisions of Copyright Law should be embraced by the courts.[8]

If the answers to these questions are no, and the courts take the position that software reverse engineering should become a proscribed act,[9] then it will introduce the intellectual equivalent of prohibition to the software industry. The problem is that in the real world, computer programmers have no choice but to reverse engineer software, their own and that of others, in order to understand what that software is really doing.

---

8. This specifically and explicitly does not mean using software reverse engineering to produce infringing copies of the original software. Surely an infringing computer program would be viewed as infringing by a court without regard to the process by which it was produced. The product is easily separable from the process that produced it to all but those opposed to software reverse engineering.

9. This position was nearly taken when the European Commission was drafting the recent software directive to harmonize copyright law within the European Community.

B. Understanding Software Reverse Engineering

There are two reasons why one needs to perform software reverse engineering. First, one needs to reverse engineer to understand how a computer program really works. Second, it is done to understand why a computer program really does *not* work. Superimposed on these two reasons are several higher level motivations such as: the desire to produce a new program that is functionally equivalent to, or better than the program under study; the desire to produce a new program that either interacts directly with the program under study or exchanges information with it; and the desire to understand why a new program fails to work in its intended environment. Thus, the motives are competition, compatibility and diagnosis.

There are only four ways to perform software reverse engineering: (1) read about the program; (2) observe the program in operation by using it on a computer; (3) perform a static examination of the individual computer instructions contained within the program; or (4) perform a dynamic examination of the individual computer instructions as the program is being run on a computer. From a technical point of view, reading the manuals, running the program, and watching what the program does, are viewed as nothing more than using the program. The programmers view static and dynamic examinations of a computer program as the only two activities which constitute reverse engineering. If one's motivation is to understand how a program works, and documentation is available, reading the documentation may provide some useful information. Documentation, by its very nature and the manner of its production, is always incomplete, inaccurate, and out-of-date when compared to the actual software itself. After all, the documentation is a statement of intent and it is merely a word picture of the program, not the program itself.

If one is motivated to reverse engineer a program because of an unexpected failure, and the intent is to understand why the program does not work, then documentation will only rarely provide the requisite information. Not surprisingly, very few software vendors describe the ways in which their programs might malfunction. Listing the problems in the manual would be regarded as bad marketing strategy. Besides, vendors do not *know* how their program will fail since if they did, it would be more cost-effective to correct the problem than to write about it.

C. Opponents Claim Reverse Engineering Is Unnecessary

Reverse engineering opponents argue that it is an unnecessary process. They claim that "any successful software product can be copied and decompiled with a flick of a console key, without significant invest-

ment or risk. Thus the decompiler can erase the lead time of the program developer and significantly reduce the originator's market for the authored work."[10] While these lines of "reason" sound true, in practice they are false. They ignore that computer programmers have routinely used reverse engineering to make up for inadequate documentation for the past 30 years.[11] Further, software thieves have yet to be seen to steal by reverse engineering. Additionally, the American software industry, even with rampant reverse engineering, leads all nations in that industry.[12]

The documentation either fails to provide sufficient details of the ideas embodied within the program, or it is completely absent.[13] Thus, if documentation is available, it will not tell why a program is failing unexpectedly. Andrew Schulman, co-author of *Undocumented DOS* and *Undocumented Windows*,[14] observes:

> [T]he problem isn't that they [Microsoft] have some Machiavellian conspiracy against the rest of the software industry, but instead that they have extremely informal approach to documentation, and to Windows itself, that is out of touch with their near-monopoly position in the industry . . . their absolutely wretched documentation is getting intolerable given their importance in the industry.[15]

A quick inspection of any worthwhile technical bookstore will reveal numerous books that augment well respected software products whose documentation is similarly wretched. All such books, to varying degrees, are borne of reverse engineering.

Observing a program in operation gives clues to what a program can do. To some degree a skilled observer can infer some general details of how the program might be working, assuming that the program is working rather than failing mysteriously. As the following examples will show, the only option guaranteed to provide accurate, complete information is to examine the software itself.

---

10. Irving Rappaport, *EC Threatens Software Protection*, SAN FRANCISCO RECORDER, Feb. 22, 1990, at 6 (Mr. Rappaport was writing as Apple's Intellectual Property Counsel).

11. This observation is based on the author's personal experience in the computer industry, which commenced in 1963.

12. Brief of Amici Curiae Computer and Business Equip. Mfrs. Assoc. at 30, Sega Enters. Ltd. v. Accolade, Inc., 977 F.2d 1510 (9th Cir. 1992) (No. 92-15655) (footnotes omitted).

13. For example, documentation is absent in the case of Nintendo and Sega home entertainment systems.

14. In this context, "DOS" refers to the Microsoft Disk Operating System, the software that allows an IBM personal computer (PC) or compatible computer to run application programs such as WordPerfect, Lotus 1-2-3 and so on. "Windows" refers to Microsoft Windows, a so-called Graphical User Interface that makes the PC's user interface appear more like that of the Apple Macintosh. Both of these very successful books are published by Addison Wesley.

15. Andrew Schulman, *Introduction* to WOODY LEONARD & VINCENT CHEN, HACKER'S GUIDE TO WORD FOR WINDOWS vii (1993).

D. Examples Of Reverse Engineering

1. An Everyday Illustration Of Competitive Reverse Engineering

With one assumption, there is a good, complete and accurate non-technical example for software reverse engineering. The assumption required, merely to make the paradigm complete, is that the recipe for a commercial food product is protectable under copyright law in addition to its protection under trade secrecy and patent law.

Hypothetical facts: Newco manufactures food products. As part of its expansion, Newco has decided to create a sauce that will compete in the same market as A1 Steak Sauce.[16] It has further decided that the way to do this is to create its own sauce that will have characteristics matching those of A1 Steak Sauce: the same color, the same viscosity, the same spicy taste, and other similar traits.

Presuming that there is no law barring the production of a "fully compatible" steak sauce, the challenge facing Newco is to learn the recipe describing the ingredients and the manufacturing process by which A1 Steak Sauce is manufactured.

What information exists in the world about A1 Steak Sauce? First, there are vast quantities of the sauce itself. Second, there doubtless is a closely guarded written recipe. Third, there are advertisements for A1 Steak Sauce, including images of the Sauce. Finally, there are probably food critics' reviews of the sauce.

The food critics' reviews, along with human taste testers and the ingredients listed on the side of the bottle, will provide Newco with subjective information about taste, texture, and general statements about possible ingredients. Professionals in the field are remarkably good at identifying individual ingredients when barraged by complex compound tastes. In all probability, Newco's own laboratories would perform sophisticated chemical analyses, perhaps using a gas chromatograph, to isolate each of the organic compounds in the sauce. This research will yield a large amount of data on the chemical composition of the A1 Steak Sauce. But when all the research data is gathered, will it provide Newco with the *specific* information on: (a) the precise ingredients in A1 Steak Sauce; or (b) a cost-effective manufacturing process to make the sauce? Clearly it will not. Merely knowing the chemical ingredients does not permit the cost-effective manufacture of steak sauce any more than one can create fine wine in the laboratory alone. Neither does it assure Newco of approval by the Food and Drug Administration (FDA).

---

16. A1 Steak Sauce is a rich brown sauce, manufactured by Brand & Co. in England and distributed in the United States by Nabisco.

For Newco to produce a commercially viable competitive steak sauce, it must employ skilled food chemists, culinary experts, and food manufacturing engineers who, by the addition of their skill and experience, can examine the research data and produce experimental versions of recipes and manufacturing processes. Only with these specialized skills, and the expenditure of a considerable amount of time, money, and effort can Newco produce a close likeness of A1 Steak Sauce.

Could Newco have reconstructed a verbatim copy of Brand & Co.'s exact recipe? Almost certainly not. There will, of course, be similarities. Both will be recipes for steak sauce. But would the same major ingredients be present in identical proportions? Would they be cooked and combined in exactly the same sequence? Would Newco's text be substantially similar to that of Brand & Co.'s text? It is very unlikely that these results would occur.

2. An Everyday Illustration Of Reverse Engineering For Compatibility

To illustrate a second important characteristic of reverse engineering in the real world, assume that the FDA received reports that some people were becoming ill after eating Newco's steak sauce in combination with Grey Poupon mustard. The FDA, based on a 100% correlation of the reports they have received, suspects that there is some adverse relationship between Newco's sauce and the mustard. The FDA, in all likelihood, would not only request access to the recipe and manufacturing process of both the steak sauce and the mustard, but would also want to test the products themselves.

It would not make sense for the FDA to examine the recipes and the manufacturing processes for the steak sauce and the mustard and then make its determination as to what the problem was based only on these materials. If the FDA found what it suspected to be the problem in the recipes, would the general public's best interest be served if they took the investigation no further? If the FDA found no apparent problem in the recipes, could it sensibly assert that the problem simply does not exist? Common sense tells us that it would make no sense at all to examine only the paperwork. The recipes and manufacturing processes are, after all, nothing more than representations of the sauce and the mustard; they are a statement of intent as to what the sauce and the mustard ought to be rather than what they actually are. They are not the sauce and the mustard themselves.

The French surrealist painter, Rene Magritte, made this point very eloquently in his image, "The use of a word," *L'usage de la parole*:

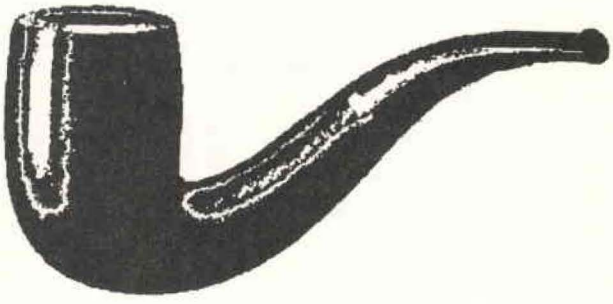


Figure 1

The literal translation of Magritte's caption is: "This is not a pipe." Indeed his painting is not a pipe. It is a painting of a pipe, a representation of an object as distinct from the object itself. It is this same distinction that must be remembered in reverse engineering: Problems only exist in the objects (or object code) themselves, not in representations of the objects. In this hypothetical situation, common sense indicates that the problem really only exists in the sauce and in the mustard. While the paperwork might give a hint as to the problem, it cannot be used as a substitute for what it represents.

To understand the precise details of the problem, the sauce and mustard manufacturers must examine their respective products. Having identified the rogue chemicals involved, the recipes might only serve to corroborate or explain the phenomena observed in the products themselves.

3. Shifting The Paradigm To Software Reverse Engineering

The preceding examples of Newco's steak sauce are an accurate model of the characteristics of software reverse engineering. To see this, imagine that Newco was a software company that wishes to produce a competitive version of Lotus 1-2-3. As before, Newco must start by considering what information might be available for study. Lotus Development Corporation will have a design or a blueprint of the overall design of 1-2-3. That corporation will also keep the human readable

source code[17] as a closely guarded trade secret. Newco should also look to the actual product as distributed into the market place. This will include user documentation that explains how the program can be used, and the object code[18] that controls the computer when 1-2-3 is loaded into the computer.

This software paradigm is a replica of the steak sauce paradigm. The steak sauce recipe is the textual representation of the sauce and corresponds directly with the computer software source code. In a very real sense, the steak sauce recipe is the "sauce code." It contains a textual description of the finished product, complete with information at a high level of abstraction, describing the ingredients, the proportions, and the process by which the steak sauce is manufactured. All of this high level of abstraction information is absent from the finished product. The actual steak sauce is the product which corresponds directly with the object code.

4. Software Reverse Engineering for Competitive Reasons

As before, Newco will not have access to Lotus 1-2-3's design documentation, nor will it have access to the software source code. The only information available to Newco is the user documentation and the actual object code that makes up the 1-2-3 program itself. To produce a competitive product, Newco must understand certain attributes of Lotus 1-2-3.[19] These attributes include: the functionality embodied in 1-2-3,[20] the user interface,[21] and the data file formats.[22]

Much of the information that Newco needs to create a design for a competitive product can be gleaned from the hundreds of pages of documentation provided when one "purchases" (technically, licenses) a retail copy of Lotus 1-2-3. The user guide describes how to make Lotus

---

17. Source code is the textual form of a computer program. It contains two ingredients, a stilted quasi-mathematical form of the instructions that will control the computer, and co-mingled with this, annotations ("comments") written by the programmer that provide high levels of abstraction information to enhance the understanding of other programmers who read the source code.

18. Object code is the computer program in a form that can actually be run on the computer. It contains the 0s and 1s in so-called "binary" notation that control the computer's operation. Object code, sometimes called "executable binary," is very difficult to comprehend even for skilled computer programmers. It contains too much information of the wrong kind to permit easy understanding. All of the high level abstraction information is removed during the process of transforming source code into object code.

19. It is a moot point from the technical perspective as to whether these attributes can be used by Newco without infringement.

20. Functionality means what the user can do with program.

21. The user interface is the external appearance and way in which the user controls what 1-2-3 does.

22. The data file formats are computer representations by which previously stored information can be input into Lotus 1-2-3, and the representations output by 1-2-3.

1-2-3 perform various calculations. The program itself also provides an experimental test-bed that can be used to verify that the program works as the documentation describes. A Newco programmer can glean significant information about the user interface by merely using Lotus 1-2-3, often deriving clues of the inner workings just by observing the program's external behavior.

External observation of the program and the documentation will provide Newco with much of the information it needs about the functionality contained within the program, with the exception of those areas where the program malfunctions (commonly called a "bug"), or where the documentation is erroneous or incomplete. The general rule, as stated before, is that all documentation for all computer software is incomplete and inaccurate. Any seasoned computer user can recount experiences where an error message appearing on the computer's display is not described in the manual, or where the manual and the computer program's behavior are at odds with each other. The program's documentation is not the program, but is a representation of the program, and of what the program should be (or what the technical writer who wrote the manual thought it would be).

Consider a hypothetical problem with the Net Present Value mathematical function in Lotus 1-2-3. Newco's programmer observes that under certain specific circumstances, it computes a result that is a few cents different from that same calculation done using Borland's Quattro Pro. Although Newco is developing a competitive product, it must now make a judgment call as to whether it should be "compatible" with this possibly erroneous calculation. Should Newco's product produce the same results as Lotus 1-2-3 even though those results are slightly in error? Experience with producing competitive computer products thus far provides a clear answer: If one wishes to compete with an existing product, there should be no measurable difference, at least insofar as such things as standard calculations, between one's own product and the existing product.

Absent any explanation in the user documentation of why 1-2-3's Net Present Value function does what it does (and not many software vendors' manuals describe how their products fail), Newco's only recourse is to examine the object code for the 1-2-3 program itself. This object code is the only entity in which the problem exists. It is the only entity available to Newco, short of industrial espionage or licensing the source code for Lotus 1-2-3. It is this examination of the object code that is the first phase of software reverse engineering.

5. Software Reverse Engineering for Compatibility

For an example of software reverse engineering for compatibility, imagine that Newco, forewarned by Atari's demise, wishes to create a

video game to run on the Super Nintendo Entertainment System (SNES) base unit. Unless the game cartridges that plug into the base unit are produced under license and manufactured by Nintendo, the games they contain will not run in the SNES base unit.

This "lockout" mechanism uses two special purpose computer chips, one in the game cartridge and one in the SNES base unit. These purpose-built central processing chips are dedicated to the specific task of interrogating each other in the manner of two jugglers tossing pseudo-random (and predictable) sequences of 0s and 1s back and forth to each other. If one electronic juggler fails to send a correct digit, fails to send it at the right instant in time, or fails to pause for the correct length of time, the other juggler, detecting the impostor, can stop the SNES base unit from running the game program in the cartridge.

Note that Newco is not motivated by the desire to produce a competitive SNES base unit, nor (necessarily) to produce game cartridges with the same types of games as Nintendo. Newco's intentions are merely to take advantage of the existing marketplace for new games. The more games that exist for a particular base unit, the more that base unit will be attractive to new buyers, so it could well be argued that Nintendo reaps some benefit from each new game available.

Given that Newco chooses to produce games for the SNES, should it be forced to sign a license agreement with Nintendo and have Nintendo manufacture the game cartridges with the special chip contained in them? Or is it legitimate that Newco could elect to divine the inner workings of the special computers and produce a computer chip capable of generating the correct mating calls? This is not the same as asking whether Newco should be able to copy the computer code within the special computer chip. All Newco's engineers need to do is understand the rules governing the pseudo-random stream of 0s and 1s, and the rules for governing the pauses that occur every so often in this data stream.

For the purposes of this hypothetical situation, let us assume that Newco elects not to become a Nintendo licensee either because it cannot afford to, or perhaps because it objects to being held to ransom.[23] In this case, what information is publicly available to tell Newco how to write a game for the Nintendo base unit (how to control the various chips in the base unit that control the display, or control the sound generator)? What publicly available information will tell them how to "unlock" the base unit by appearing to be a licensed game cartridge?

---

23. Would Newco have to sign a license to write a game to run on the IBM personal computer, for example?

The answer to both questions is none. There are no published manuals, books, or magazine articles describing any of the internal details of the SNES base unit. Neither is there any publicly available information on how the two special computers in the base unit and the game cartridge respectively perform their mating calls.

Newco's only recourse, absent a license with Nintendo, is to analyze the following components of the SNES system: the base unit's hardware and software; a game cartridge's software; and the special computer chips used to lock out game cartridges not manufactured by Nintendo. By this hardware reverse engineering, Newco can divine the following: how to write games that will run on the SNES base unit, and how to emulate the behavior of the lockout chip on the game cartridge so that the base unit will permit the game program to run.

Opponents of reverse engineering have, in the past, asserted that there was no need to reverse engineer because all the information needed for compatibility could be obtained by observing a program in operation and by studying the available documentation.[24] These opponents tend to fall silent when confronted by the previous hypothetical situation since there is no documentation whatsoever, and the lockout chips operate invisibly, producing no output to be observed. Even when an electronic device is attached to the communication lines between the two lockout chips, it would only reveal the intermittent bursts of pseudo-random 0s and 1s of pseudo-random length being transmitted at pseudo-random intervals.

Ignoring how Newco *should* react, in the real world Newco would be left confronting two problems: (1) how to write games for the base unit; and (2) how to make a lockout chip that will be accepted by the lockout chip in the base unit. Newco's engineers, observing the data stream between the lockout chips, would probably conclude that some quite sophisticated process was being used by the chips both to generate the data bursts, and to control the lengths of these bursts and the lengths of the silences between data bursts. It would not take many hours to realize that the only sure method to understand the underlying ideas behind the lockout chip's algorithms would be to reverse engineer the actual code. Merely observing the data stream itself is nearly useless and very time-consuming. How long should one observe it? 100 hours? 200 hours? What if after 201 hours or on some specific date the lockout chips are designed to switch to a different algorithm? Clearly, Newco's engineers would not be able to divine the correct length of time during which the observations must be sustained. Merely using the data stream to divine the underlying algorithm is also error prone.

---

24. Rappaport, *supra* note 10, at 6.

Can one really discern with certainty an underlying process merely by observing the output? Accuracy increases the longer one continues to observe, and the question then reverts to: How long is enough? An examination of the code in the lockout chips, although it might be time consuming and laborious, is unequivocal. The examination will define completely and precisely the rules being used to generate the data stream, whether for 100 milliseconds or 100 years.

Learning from the error of Atari's ways, or more specifically, the error of Atari's then current outside attorneys' ways, Newco would know better than to attempt to get the deposit copy of Nintendo's lockout chip source code from the Copyright Office under the guise that it was required for litigation.[25] Instead, Newco would again be forced to take the more costly and laborious route of physically reverse engineering the lockout chips, removing the protective plastic that encapsulates the chip, and creating photomicrographs of the read-only memory containing the mating call data-generating code. In many special-purpose microcomputers, the 0s and 1s that make up the program are not laid out in neatly serried ranks waiting for the reverse engineer to happen by. Usually, they are stored in a "scrambled" form, either for reasons of engineering or manufacturing simplicity, or in some cases, to discourage reverse engineering.

As the lockout chips are probably special purpose computers, Newco engineers would not be able to make any sense of the 0s and 1s that make up the mating call program until they had also reverse engineered the computer chip itself. Is an ADD instruction 0011 or 1010? It all depends on how the central processing unit that forms the heart of the computer chip has been designed. Only then would Newco's engineers be able to examine the object code for the mating call data generation program and truly understand how they could create their own computer and code to generate appropriate mating call data. Of course, Newco would have to be particularly careful to ensure that the resulting computer chip and computer code was not substantially similar to the Nintendo lockout chip and computer code. Infringement is still infringement regardless of the process used to produce the final product.

Newco's engineers must also reverse engineer the Nintendo base unit. Observation of the silk-screened notations visible on some of the integrated circuits might give the engineers some clue as to their functions. Careful observation of the printed circuit board's wiring will add more clues. But many of the more complex special-purpose chips will

25. As was the case in *Atari Games Corp. v. Nintendo of America, Inc.*, 975 F.2d 832 (Fed. Cir. 1992).

either have no markings, or will bear only proprietary markings that mean nothing to the outside world.

Again, Newco's engineers will realize fairly quickly that the only certain method of divining how to write a game is to examine the object code contained within existing games. By examining other games, they will be able to see how the games initialize the various bits and pieces of hardware in the SNES, and how they make graphic images dance across the screen and play the corresponding music.

This hypothetical situation is not an extreme case. In other contexts, there might be some documentation available. Whatever documentation is available, however, can be guaranteed to be inaccurate, incomplete and out-of-date. Therefore, the situation quickly reverts to match the "no documentation at all" scenario described above. The only entity that will provide certain and accurate knowledge of the requirements for compatibility is the software and/or the hardware itself.

E. Software Forward Engineering

To truly appreciate both the difficulty of software reverse engineering and the flaws in reasoning exhibited by some of those who oppose it, it is necessary to have some understanding of the process of software development ("forward" engineering) as a foundation for reversing the process. Software development consists of several phases, although not usually as well defined in the real world as the following paragraphs might imply.

1. The Specification

A software designer creates a specification embodying all of the ideas that constitute the program to be developed. Embedded in the specification are all of the higher levels of abstraction information. This information includes the reasons for creating the program, the requirements of time and space, and the general algorithms that must be performed by the program.

2. The Source Code

This specification is handed to a programmer who creates "source code" for the program. This source code is a human readable form of the program, written in a procedural artificial language invented specifically for stating what a computer must do to solve a problem. By careful choice of symbolic names for various objects within the program, a programmer can give vitally important clues of his or her intentions to anyone who reads the source code. For example, the statement in the "C" language:

```
if ((flag1 == 1) && (flag2 == 1))
      {
      just_do_it();
      }
```

snaps into chilling reality when one is told that: (a) this code example may be found in a giant program running in a giant computer buried under a mountain in Wyoming (at Norad's HQ); (b) this program tests two conditions and if both are true, decides to do something; and (c) with only minor changes that add in higher-level of abstraction information the code appears:

```
if ((incoming_missiles == 1) && (presidential_approval == 1))
      {
      launch_retaliatory_strike();
      }
```

Programmers also embed many lines of so-called "comments." This commentary, which plays no part in the guidance of the computer, is merely text interleaved between language that guides the computer. It is the equivalent of marginal annotations and is intended to assist the original programmer or those that follow in understanding why the program was crafted in a particular way, or to explain a particularly complex flow of logic. There are no restrictions on what must or must not be written in comments, but inevitably they are the repository of all the knowledge that the programmer has in his or her head as the code is being created. One also frequently sees a certain irreverence in the commentary which is a by-product of the exuberance of programmers and is best not taken too seriously, as illustrated by the following example:

```
/* The following code tests whether or not to start World War IV
(World War III was in the Persian Gulf). If this code ever gets
executed, put your head between your knees and kiss yourself goodbye.
By the way, the incoming_missiles variable should have been set to
something meaningful before the CPU gets to this point.
At least I think it is....let me see....oooohhhh....maybe not...

if ((incoming_missiles == TRUE) && (presidential_approval == TRUE))
      {
      launch_retaliatory_strike();
      }
```

The essence of these comments is that they contain higher level information included specifically to help understand what was going through the programmer's mind as he or she wrote the code. Bearing in mind that it costs approximately ten times more to maintain a program during its useful life than it cost to develop in the first place, it is easy to

understand why a maintenance programmer needs all the help he or she can get.

In contrast to the "high level" programming language example above, programmers can also write programs with very detailed, low-level instructions to the computer. This can be illustrated by pondering how to tell someone, via a telephone line, how to tie a bow in a shoelace. Each instruction in this so-called "assembly language" must be very detailed; each line of source code, if it is not a comment line, corresponds to one instruction to the central processing unit (CPU) that actually manipulates data within the computer.

An example of assembly language programming quickly reveals the detailed level at which the programmer must operate:

```
RETALIATE:
      MOV   AX,INCOMING ;Get incoming missiles flag into AX register
      MOV   BX,PRESAPP  ;Get presidential appr. into BX register
      AND   AX,BX       ;Boolean AND of two conditions
      JNZ   LAUNCHEM    ;Let's go do some damage
```

The word "RETALIATE:" is a symbol label used elsewhere by the programmer to reference this code. It is equivalent to a paragraph heading in a document.

The strange abbreviations like "MOV," "AND," and "JNZ" are mnemonic names given to each of the instructions that the computer is capable of executing. A desktop calculator has mnemonic names with "+" meaning add, "-" meaning subtract and so on. In this example, "MOV" means "move data from the second thing to the first thing," (i.e., "MOV AX, INCOMING" means "move the contents of the data storage area called INCOMING into the AX register[26] of the CPU so that it can be manipulated"). "AND" performs a logical "anding" of the current contents of registers AX and BX, leaving the results in AX, and "JNZ" means "Jump out of sequence to a symbol label named LAUNCHEM if the result of the AND was non-zero," (i.e., if there are incoming missiles AND the president has approved a counter-strike).

3. The Object Code

A computer cannot run a program in source code form. The source code must be translated from text into a form that contains instructions

---

26. A register is a temporary storage area inside the computer's central processing unit. The actual display or the "memory" of a desktop calculator are examples of a register. A number must be visible in the display or stored into the calculator's memory before it can be used as a part of the calculation. The design of each central processing unit determines how many registers it will have and by what abbreviated names they are known: "AX," "BX," and so on.

to the computer known as object code. Object code consists of numeric codes specifying each of the computer instructions that must be executed, as well as the locations in memory of the data on which the instructions are to operate. A special program is used to translate the human-readable source code into computer-readable object code. If the program has been written in a high-level language like C, the program is called a compiler. If the program has been written in a low-level language, then it is called an assembler.

Compilers are very sophisticated programs. Not only do they translate the source code into object code, they also optimize it by re-arranging the low-level instructions to make the program run faster. An analogy of optimization would be re-arranging a shopping list into the order in which one would go to the various stores, rather than the order in which one first thought of the items. The list is easier to read that way and it takes a lot less time to purchase all of the items if one does not have to go backwards and forwards between different stores.

The object code output by compilers and assemblers has several characteristics in common. First, all of the comments have been stripped out of the program as they are irrelevant to the computer. The computer could make no sense of these strings of characters anyway. Second, all of the symbolic names have been stripped out of the program. They, too, have absolutely no meaning to the computer. Finally, additional chunks of object code have been appended to the object code resulting from the source code. This additional object code, stored in previously prepared libraries of object code, is "helper" code designed to make the program's (and the programmer's) task easier by obviating the need to write source code for the repetitive tasks that all programs have to do (such as preparing data files for processing, sending messages to the screen, reading data from the keyboard, and so forth).

All that remains in the object code file are the numeric codes that represent the instructions to be executed, and the numeric codes that represent the data locations in memory on which the instructions must operate when the program is run. The object code file, as mentioned above, is a composite of the instructions written by the programmer plus the object code brought in from libraries of prefabricated object code.

4. Running a Program on a Computer

Once a program has been translated into object code it can be run on the computer. Running the program means that a copy of the object code file on the computer's disk must be made in the computer's main memory. Once this copy has been made, the central processing unit is told at what location in memory it must start executing instructions. From that point on, the central processor executes one instruction after

another from the program until it reaches a point in the program where it encounters an instruction that tells it that the program has completed.[27]

F. Reverse Engineering a Program: A Worked Example

To illustrate the process of reverse engineering consider a hypothetical example of a program written by a programmer at Sensatemp Incorporated. Sensatemp makes temperature sensors using microcomputers to compensate for any errors caused by the physics of the sensors themselves.

Newco is a competitor of Sensatemp and wishes to study the part of Sensatemp's device that is responsible for reading temperatures in Fahrenheit and converting them over to Celsius, compensating for instrument errors according to the actual Celsius temperature measured. For these purposes, Newco's engineers have isolated the portion of the firmware (that is, software embodied into a Read Only Memory (ROM) chip) that performs this function. They now wish to reverse engineer it to find out exactly how Sensatemp's instrument works. Newco's engineers want to develop a functionally equivalent version for compensation so that their instruments offer similar capabilities.

1. Possible Reverse Engineering Strategies

There are three strategies available to Newco's engineers: (1) Read the Manuals: Read all of the available documentation to find out what the company's technical writers say about the program and how it ought to work; (2) "Black Box" Observation: Use the Sensatemp instrument and observe what it does and attempt to infer what must be going on inside the program; and (3) Reverse Engineer: Study the object code statically, and run the program in an experimental environment that permits observation of the inner workings of the program as it executes.

a. Read the Manuals

Although stated previously, it bears repeating that documentation is always incomplete, inaccurate, and out-of-date. It cannot help but be this way as the documentation is merely a statement of how the actual computer program *should* be, rather than how it *is*. Therefore, whenever some unexplained behavior occurs in a program being reverse engineered, the odds are that the manuals will not be much help.

---

27. This is an oversimplification. There are many other things that may occur to stop a program's execution, but for clarity they have been ignored.

In this hypothetical example, Newco's engineers discover that the documentation for the Sensatemp product states (not unreasonably) that: "The Sensatemp XR1774-10Q probe has been compensated to correct for any errors caused by the non-linearity of the sensing element. This compensation takes the form of an appropriate percentage correction (in the range 1%-4%) on the final temperature in Celsius displayed by the unit."

Armed with this additional information, Newco's engineers surmise that the differences between the theoretical calculations and the actual results are a result of this compensation. But that still does not explain everything. If the sensing unit is reading a temperature of 100°F, that still cannot explain why the conversion is "inaccurate." Could it be that when the sensor unit is reading some number just above 100°F that it erroneously indicates the temperature is 100°F? The "inaccuracy" could therefore be the compensation being applied. Comparing theoretical calculations with the printout again: 0°F is -17.7777°C not the -17.9556 shown (a difference of -0.1778); 100°F is 37.7777°C not the 38.1556 shown (a difference of 0.3778). The differences are almost exactly 1% of the observed temperature. Newco's engineers might therefore reasonably assume that a correction factor has been applied. For completeness, Newco's engineers check the theoretical conversion for 200°F, obtaining a value of 93.3333°C. This is precisely the value shown by the Sensatemp instrument, which begs the question why should this be so? Is any correction being applied? It does not appear to have been applied to the 200°F temperature. Checking all the other values, Newco's engineers discover that any temperature less than 48.8888°C appears to be wrong by 1% of the temperature expressed in Celsius degrees. What can this mean? Are they seeing temperature compensation? Or is it something else?

The printout from observing the program does not tell Newco's engineers what they need to know. The documentation told them what should be happening, and if their observations and calculations are valid, even they do not contain the whole truth. At this point, Newco's engineers' only recourse is to reverse engineer Sensatemp's program. It is only the actual Sensatemp program that can and will tell them what is going on. There is just not enough information available to them from their observations of the program or reading the documentation.

b. "Black Box" Observation

When Newco's engineers use the instrument to measure carefully preset temperatures in Fahrenheit, the Sensatemp box displays:

```
Temp. F =   0.0000, C = -17.9556
Temp. F =  20.0000, C =  -6.7333
Temp. F =  40.0000, C =   4.4889
Temp. F =  60.0000, C =  15.7111
Temp. F =  80.0000, C =  26.9333
Temp. F = 100.0000, C =  38.1556
Temp. F = 120.0000, C =  48.8889
Temp. F = 140.0000, C =  60.0000
Temp. F = 160.0000, C =  71.1111
Temp. F = 180.0000, C =  82.2222
Temp. F = 200.0000, C =  93.3333
```

This is essentially a conversion table, with what appears to be temperatures in degrees Fahrenheit and Celsius. To the untrained eye, this output may not reveal much, but Newco's engineers can infer several things from it: (1) Based on the number of decimal places shown in the printout, the program is apparently using an internal representation of numbers known in the scientific community as "floating point;"[28] and (2) the temperature probe appears capable of sensing temperatures between 0°F and 200°F.

Do Newco's engineers know this information for a fact? Absolutely not. They are making reasonable guesses based on their observations. In fact, reverse engineers spend most of their time guessing, and using words like "What?," "Why?," and "Where?". After some minutes staring at this printout, a small voice in the back of one of the Newco's engineer's heads might say "Check the conversion!" This done, the Newco engineers will discover that the numbers displayed do not appear correct. For example: 0°F is -17.7777°C not the -17.9556 shown, and 100°F is 37.7777°C not the 38.1556 shown.

What is going on here? Why are the conversions mathematically inaccurate? One really cannot say. Could it be that Sensatemp's programmers used a slightly different formula for conversion, or is there some other possible explanation? Without further information, inspiration, or blind luck, Newco's engineers cannot establish the precise cause for the apparent errors.

c. Reverse Engineering: Static Examination

As a prelude to reverse engineering Sensatemp's program, Newco's engineers open up the Sensatemp instrument. Inside they find a printed circuit board with about a dozen integrated circuit chips

28. It is called the "floating point" because the decimal point can float either left or right to permit the computer to represent either very large or very small numbers.

mounted on it. Some of these chips are marked with manufacturers' code numbers, but several are unmarked.

Where is the actual program to be reverse engineered? What type of central processor unit will run the Sensatemp program? Newco's engineers start from a position of ignorance. They must first examine the physical details of the printed circuit board, look up data sheets for those chips that are marked, and attempt to infer additional information about the unmarked chips.[29]

By visual examination, augmented by some simple electrical continuity testing, Newco's engineers can determine, at least at a superficial level, the electrical connections on the Sensatemp board. From this they can divine what each chip's function might be and can tentatively determine which chips might contain read only memory, and are therefore candidates for further examination.

There are only two methods for examining the contents of a ROM chip: electronically or physically. The electronic technique can be implemented in two ways depending on whether or not the ROM chip can be removed from the printed circuit board. If the chip can be removed, it can be placed into a test rig and its contents read out directly as a set of 0s and 1s. If, however, the ROM chip cannot be removed from the board without damaging it and preventing its contents from being read out, Newco's engineers will need to devise a mechanism for "looking inside" the Sensatemp unit while it is actually switched on and operating. Depending on the central processing unit chip used by Sensatemp, Newco's engineers might be able to use a commercially available piece of electronic test equipment called an "in-circuit emulator" (ICE). An ICE replaces the central processor chip with electronic equipment that "emulates" the exact electronic behavior of the CPU. The printed circuit board works exactly as before, with the exception that now, by means of the ICE, an outside observer can monitor exactly what is happening inside the computer system. The ICE can also be used to read out an image of the ROM, reading out the 0s and 1s as though the ROM chip had itself been removed.

The physical technique involves "deprocessing" the ROM chip by removing the outer layer of protective plastic and exposing the small silicon chip itself. Acid is used to etch away some of the outermost layers of the chemicals deposited on the silicon chip and photomicrographs can be taken of the circuitry that makes up the ROM itself. At suitable magnification, a trained engineer can discern the actual 0s and 1s stored in the ROM chip. It is not unusual, however, for engineers to

29. Being unmarked is one indication that Sensatemp might want to make it harder for engineers such as Newco's to divine each chip's purpose.

rearrange the individual binary digits within a ROM chip to make it easier to manufacture the chip, thereby "scrambling" the physical arrangement that will be seen in the photomicrograph. Newco's engineers will also have to decode some of the chip's surrounding electronics in order to relate the physical layout of the binary digits to their correct electronic layout. Only then can they read the ROM chip in the same way that the central processor chip will.

At this point, both the electronic and physical methods converge. Both have now yielded an image of the ROM's contents, the actual 0s and 1s presented to the central processor unit. The task facing Newco's engineers is now to decode these 0s and 1s and divine what instructions and what data will be presented to the CPU.

2. Examining a Program in the Computer's Memory

Internally, the computer uses a numeric representation that is even simpler than decimal, the "binary" system. In binary, all numbers are represented by one of two digits, 0 or 1. Unlike the familiar column headings of units, tens, hundreds, thousands, and so on, the binary system uses 1, 2, 4, 8, 16, 32, and so on. All of the same rules for doing decimal arithmetic still apply, the only difference is that binary is based on 2 rather than 10. For example, the decimal number 123 (which represents 1 hundred, 2 tens and 3 ones), will appear as 1111011 in binary. This represents 1 "64", 1 "32", 1 "16", 1 "8", 1 "2" and 1 "1", which totals to 123 in decimal. Seeing binary with column headers helps:

64	32	16	8	4	2	1
1	1	1	1	0	1	1

Computer memory is most accurately thought of as a giant pigeonhole storage area. Visualize the front desk of a large hotel behind which there is a pigeonhole for each room. Each pigeonhole has a unique number corresponding to the room number, and a small storage area. So, in a large Las Vegas hotel with a thousand rooms, one might see a thousand pigeonholes, numbered from 1 to 1000.

In a computer system two things are different. First, the "pigeonholes" start numbering from 0, thereby making the electronics easier. Second, each pigeonhole can store a very limited amount of information. Basically, the number must range between 0 and 255 decimal formed by grouping together eight binary digits. One value in the range 0 to 255 is all each memory location (pigeonhole) can ever store at any one moment in time. By technical sleight of hand, computer designers can make it appear as though the computer can store a character of the alphabet or can group adjacent pigeonholes together to make the computer deal with numbers larger than 255, or store a negative number.

Memory locations appear to store characters of the alphabet using a simple encoding system (school children sometimes use such a scheme at school to pass encrypted messages in the class room). A simple example of this encoding is to say that the number 1 represents "A," the number 2 represents "B", and so on. A computer system is designed to convert incoming keystrokes into their appropriate number, and convert the numbers back into their appropriate letters when it displays them on the screen, thereby creating the illusion that it is operating on alphabetic characters.

In practice, "A" does not equal 1, but 65. In binary, this would appear as 1000001. Other characters were assigned to the first 65 numbers, 0 to 64. These include "non-graphic" characters that are not visible on the computer screen, such as Carriage Return, Tab, and so on. The entire upper case alphabet, the lower case alphabet, and the special "mark" characters such as parenthesis, and percent sign then take up the remaining numbers up to 255 (0 to 255 represents a total of 256 numbers). This encoding system is recognized around the world by the acronym given it by the American National Standards Institute, ASCII, standing for "American Standard Code for Information Interchange."

In the computer's memory, these binary digits are stored in groups of eight. Binary digits are abbreviated to "bits," and these groups of eight are known as "bytes." On occasion, programmers will group these bits into groups of four, known as "nybbles."[30] To identify each byte in memory, each is assigned an "address." This address is just a number. It starts at zero and increases by one, rather like the numbers on the mail pigeonholes behind the hotel registration desk (except that in the hotel there is no room number 0).

a. Looking at Binary

Opponents of reverse engineering (almost always either lawyers or representatives of large software companies or lawyers who represent large software companies), claim that to avoid making an allegedly infringing intermediary copy of a program, those who wish to reverse engineer software must only look at the binary data as it is stored in the computer's memory. Their argument is that loading into memory does not constitute a permanent fixation of the data, therefore the copy made in the computer's memory is not an infringing copy. This argument was advanced by Sega in *Sega Enterprises Ltd. v. Accolade,*

30. There is controversy in the computer industry as to whether the correct spelling is "nibbles" or "nybbles."

Inc.,[31] and accepted initially by the lower court. Here, therefore, are the first few lines of Sensatemp's program as they are stored inside the computer:

0000000000000010000000010000101100000000000000000100000000000000
0000000000000000001000000000000000000000000000000000111011111100
0000000000000000000011010000100000000000000000000010000000100000
0000000000000000000000000000000000000000000000000000000000000000
0010010000010111010001111110111100000000000001000010001000000010
1110010110000001010010011111001100011000000010000100011110011100
0000000000000010000000000000000001001000010101000100100001010011
0010111100000010010011011111100000000000000000000011000010010011 0
0100111010111001000000000000000000100000100110000100111010111001
0000000000000000001000001010000011011110111111000000000000001100
0010111100111100000000000000000000000000000000000100111010111001
0000000000000000001111011001000001011000100011110010111100000000
0100111010111001000000000000000001111011101110001100000001011 10
0100000000101000001000110010100101100011011100100111010000110000
0010111001110011000010010011000100101110001100100000100100111000
0011011000101111001100010011000000101111001100000011011100001001
0100001101101111011100000111100101110010001000000011000100111001
0011100000110101001000000101001101110101011011100010000001001101
0110100101100011011100100110111100000000000000000100111001110101
0100111011111100100000000000000000010101110010000000000000000000
0100111001010110000000000000000011011111111111001111111111111111
1111111111110000010010001101011100000000000000000010110101111100
0000000000000000000000000000000011111111111111000010000000101110
1111111111111100010011101011100100000000000000000010101011111100
0010110101111010000000010011111011111111111100000010110101111010
0000000100100100001000000010111011111111111111000100111010111001
0000000000000000001010101111110001000001111110100000000100100010
0100111010111001000000000000000001001101100010001001110101110 01

. . . and so on. There are a total of four thousand lines of output in this printout, fifty pages of unrelenting 0s and 1s.

Not even programmers who have been programming since the days of vacuum tube computers look at computer programs in this form, and for good reason. It is completely impractical to look at more than just a few lines of binary.

---

31. 977 F.2d 1510 (9th Cir. 1992).

As a sneak preview, Sensatemp's program for this hypothetical example consists of just eight source code lines. Not eight hundred, nor eight thousand, but just eight active lines of source code. The entire source code, including blank lines and commentary, totals 917 characters. The object code produced by the compiler, when combined with the necessary modules from object code libraries, totals 32,768 bytes, all for just eight active lines of code.

Clearly, this form of the data is impractical. If an eight line "toy" program creates a fifty page document of raw binary data, a real-world program might be 500,000 bytes long. If displayed in raw binary this would require 62,500 lines of printout on a total of 1,250 pages—a stack of paper about six inches thick. It is argued that such a printout would be an infringing copy and that any handwritten annotations that Newco's programmers might write on the printout were embellishments in what would finally be a derivative copy of the original binary form of the Sensatemp program. Newco's programmers may well have to commit the remainder of their born days trying to understand the binary patterns in raw binary. Those who propose that programmers merely display the raw binary of a computer program have absolutely no understanding of the real world. Such a proposal is equivalent to suggesting that, as nail scissors can cut a blade of grass, all lawn mowers should be abolished and scissors used instead.

To understand a binary image, Newco's programmers must convert it into a human comprehensible form. Note that this is merely a conversion, no new information is added to the underlying binary image. Certainly no high level information from Sensatemp's original source code can be added, as it is absent from the binary image. The first of several conversions will be to group the binary digits into 8-bit bytes like this:

```
00000000 00000010 00000001 00001011 00000000 00000000 01000000 00000000
00000000 00000000 00100000 00000000 00000000 00000000 00001110 11111100
00000000 00000000 00000110 10000100 00000000 00000000 00100000 00100000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00100100 00010111 01000111 11101111 00000000 00000100 00100010 00000010
11100101 10000001 01001001 11110011 00011000 00000100 00100011 11001100
00000000 00000010 00000000 00000000 01001000 01010100 01001000 01010011
00101111 00000010 01001101 11111000 00000000 00000000 01100001 00100110
01001110 10111001 00000000 00000000 00100000 10011000 01001110 10111001
00000000 00000000 00100000 10100000 11011110 11111100 00000000 00001100
```

This step improves Newco's engineer's ability to perceive the contents of the computer's memory. It is, however, still almost impossible to know where on the screen the byte at, for example, location 25 might be. The example below starts at location 7193:

```
01100000 11000100 00001100 10000000 00110110 10010000 00000000 00000000
01101100 00000100 01000010 10000000 01100000 10111000 11100011 10001001
11100011 10010000 11100011 10001001 11100011 10010000 11100011 10001001
11100011 10010000 01001010 10000001 01100111 00001000 00000000 10000001
10000000 00000000 00000000 00000000 01100000 00000010 01000010 10000001
01001000 01000000 00110010 00000000 01001000 01000000 11101110 01001001
00000010 10000000 00000000 01111111 11111111 11111111 00000000 10000000
00000000 10000000 00000000 00000000 00000100 01000001 00000001 10000000
01101100 00010000 11100010 10001000 01100100 00000000 00000000 00001000
00000000 10000001 10000000 00000000 00000000 00000000 01010010 01000001
01101101 11110000 01010010 10000000 00001000 00000000 00000000 00000000
01100110 00001100 00001000 00000001 00000000 00011111 01100110 00000110
00000010 10000000 00000001 11111111 11111111 11111100 11100010 10001000
01100000 00000000 11111111 01011100 01001000 11100111 00111111 00000000
```

To be practical, Newco's engineers must be able to display memory in such a way that they can see both the instructions to the computer and the memory locations contained in those instructions, plus any embedded ASCII characters.

Rather than displaying the contents of memory in binary, where there is far too much information in the wrong format, Newco's engineers could use a compressed form of binary called hexadecimal. Hexadecimal is base 16 arithmetic (from the Greek and Latin, hex and decim). To convert to hexadecimal, binary digits are grouped together in groups of four and placed under the normal binary headings like this:

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A (Since '9' is the last digit, letters are used instead)
1011	B
1100	C
1011	D
1110	E
1111	F

Hexadecimal notation demands "numbers" to represent the values that correspond to the decimal values 10, 11, 12, 13, 14, and 15 and so the first few letters of the alphabet are pressed into service. This can lead

to some ambiguity. For example, is "AB" the first two letters of the alphabet or the binary pattern 10101011? One cannot tell unless there is some indication of whether the value is stated as two ASCII characters (occupying two adjacent bytes), or two hexadecimal numbers (one for each nybble in a byte). Programmers are always careful to indicate when numbers are stated in hexadecimal (abbreviated to "hex" for short). Numbers may be written either as 1FH, 1fH or even 0x1F, all of which denote that the 1F is to be taken in hexadecimal. Hex representation has the advantage that it effectively compresses the amount of space one needs to display binary values, as each hex digit represents four binary digits.

b. Looking at Hexadecimal and ASCII

Armed with this new hexadecimal notation and the knowledge that the computer can also represent letters of the alphabet using the ASCII coding system, Newco's engineers can now look at the binary image of Sensatemp's program in a new light:

```
0000:  00 02 01 0b 00 00 40 00   00 00 20 00 00 00 0e fc   ......@.  .. ....|
0010:  00 00 06 84 00 00 20 20   00 00 00 00 00 00 00 00   ......    ........
0020:  24 17 47 ef 00 04 22 02   e5 81 49 f3 18 04 23 cc   $.Go..".  e.Is..#L
0030:  00 02 00 00 48 54 48 53   2f 02 4d f8 00 00 61 26   ....HTHS  /.Mx..a&
0040:  4e b9 00 00 20 98 4e b9   00 00 20 a0 de fc 00 0c   N9.. .N9  ..  ^|..
0050:  2f 3c 00 00 00 00 4e b9   00 00 3d 90 58 8f 2f 00   /<....N9  ..=.X./.
0060:  4e b9 00 00 3d dc 60 2e   40 28 23 29 63 72 74 30   N9..=\`.  @(#)crt0
0070:  2e 73 09 31 2e 32 09 38   36 2f 31 30 2f 30 37 09   .s.1.2.8  6/10/07.
0080:  43 6f 70 79 72 20 31 39   38 35 20 53 75 6e 20 4d   Copyr 19  85 Sun M
0090:  69 63 72 6f 00 00 4e 75   4e f9 00 00 2b 90 00 00   icro..Nu  Ny..+...
00a0:  4e 56 00 00 df fc ff ff   ff f0 48 d7 00 00 2d 7c   NV.._|..  .pHW..-|
00b0:  00 00 00 00 ff fc 20 2e   ff fc 4e b9 00 00 2a fc   ......|.  .|N9..*|
00c0:  2d 7a 01 3e ff f0 2d 7a   01 3c ff f4 41 ee ff f0   -z.>.p-z  .<.tAn.p
00d0:  4e b9 00 00 26 44 62 00   01 24 20 2e ff fc 4e b9   N9..&Db.  .$ ..|N9
00e0:  00 00 2a fc 41 fa 01 22   4e b9 00 00 26 c4 4e b9   ..*|Az."  N9..&DN9
00f0:  00 00 22 38 2d 40 ff f8   20 2e ff f8 4e b9 00 00   .."8-@.x  ..xN9..
0100:  2a fc 41 fa 01 0c 4e b9   00 00 24 80 4e b9 00 00   *|Az..N9  ..$.N9..
0110:  22 38 2d 40 ff f8 20 2e   ff f8 4e b9 00 00 2a fc   "8-@.x .  .xN9..*|
0120:  2d 7a 00 f6 ff f0 2d 7a   00 f4 ff f4 41 ee ff f0   -z.v.p-z  .t.tAn.p
0130:  4e b9 00 00 26 44 62 3c   20 2e ff f8 4e b9 00 00   N9..&Db<  ..xN9..
```

In the example of a so-called "hex dump" above, the memory address of the first byte on the line is shown first. This address is also shown in hexadecimal. Then come two groups of eight bytes, with each byte's value being shown in hexadecimal (the grouping is just to make it easier to count off across the line). On the far right, again grouped in two groups of 8, comes the same data, but this time displayed as ASCII characters. Those bytes that have values that do not correspond to "visible" characters like letters and numbers (such as Carriage Return and Tab) are shown as a period.

Notice the lines that start 0070, 0080 and 0090. On the right hand side the bytes are shown as characters in the ASCII character set and

spell out the message "crt0.s 1.2.8 6/10/07 Copyr 1985 Sun Micro." With one or two exceptions, all of the other strange characters on the right are artifacts of viewing non-ASCII characters as though they were ASCII.

This fragment of the Sensatemp program appears to be some object code that has an embedded copyright message in it. This is Sensatemp's program, yet there is a copyright message in it for Sun Microsystems. This would tell Newco's engineers that they are looking at the output of Sun's C compiler, and that Sensatemp's code has been glued together with some helper code written by Sun. Although not too terribly helpful to Newco's engineers in their quest for understanding the Sensatemp code, it does illustrate the peril of blindly sending the first and last twenty-five pages of a hex dump to the Copyright Office as a deposit copy. One may be faithfully sending the Register of Copyrights fifty pages of some other company's object code!

Although this hexadecimal dump changes the appearance of the information Newco's engineers saw in their original binary dump of the program, it has not added any new information. Only the representation has been changed. Nothing has been added to or subtracted from the information in the original binary dump.

c. Need We Go Beyond A Hexadecimal Dump?

The so-called "hex dump" described above is the first representation of the computer program's object code that approaches something that can be understood by programmers, which, of course, is why programmers create them. Because of this, opponents of reverse engineering argue that a reverse engineer has no need to proceed any further. Opponents assert that programmers can read object code directly.[32] As any experienced programmer can report, back in the mid-to-late-1960s, programmers would debug their programs by looking at hex dumps because there was no other way.

Newco's programmers, not being of the generation when programmers had to memorize the CPU's binary instruction codes, would be aghast at the complete impracticality of working from hexadecimal dumps. While it would be possible for them to read the hexadecimal dumps and decode them, it is impossible (consistent with industry deadlines and their lifetime) for them to read and understand the program in sufficient detail.

---

32. Anthony L. Clapes, *Confessions of an Amicus Curiae: Technophobia, Law and Creativity in the Digital Arts*, 19 U. DAYTON L. REV. 903 (1994).

d. Looking at CPU Instructions

For Newco's engineers to be able to understand what the Sensatemp program is doing, they must be able to examine how the program is functioning. But where are the computer instructions? How can they separate instructions from embedded text messages or other data values used by the program? The answer is, only with great difficulty. Instructions to the CPU are nothing more than particular binary values. The only way that the CPU "knows" that a given set of bytes is an instruction is when the CPU is directed to execute it! Until that moment, it is difficult to know whether the bytes contain an instruction or data.

While this might seem confusing, there are other instances of this in ordinary life. For example, if one sees the words "Long may she reign over us," can one tell whether this is prose to be read, or the words of a song to be sung? One cannot tell. There is no contextual information to indicate the correct choice. On the other hand, if one were asked to sing these words one could surmise they were part of a song. The same logic is true for the computer. The contents of memory can be either instructions or data. If the CPU is directed to operate on a particular location as though it were data, then it is data. Alternatively, if the CPU is directed to operate on a particular location as though it were an instruction then it had better be an instruction, otherwise the computer will behave in an unpredictable way. It is the programmer's responsibility to ensure that the CPU is always presented with valid instructions.

As Newco's engineers look at a hex dump, they have two problems: (1) finding the first instruction in the program; and (2) using that as the starting point for a journey through the instructions in the program. They know that the computer, unless directed otherwise by "branch" or "jump" instructions, will always execute adjacent instructions one after the other. Therefore, Newco's engineers must be vigilant and make a note of each branch in the maze of instructions, unwinding a mental ball of thread so they can retrace their steps to the previous branch and thereby explore both the "left" and "right" turns at each branch in the maze.

To make a long example shorter, assume that Newco's engineers know that, for this CPU, the first instruction of the program occurs at location 00a0 in the hexadecimal listing. Here is the relevant fragment of the hex dump again with the first few bytes of Sensatemp's program shown in bold italics:

```
0080:  43 6f 70 79 72 20 31 39  38 35 20 53 75 6e 20 4d   Copyr 19  85 Sun M
0090:  69 63 72 6f 00 00 4e 75  4e f9 00 00 2b 90 00 00   icro..Nu  Ny..+...
00a0:  4e 56 00 00 df fc ff ff  ff f0 48 d7 00 00 2d 7c   NV..._|..  .pHW..-|
00b0:  00 00 00 00 ff fc 20 2e  ff fc 4e b9 00 00 2a fc   .....| .  .|N9..*|
00c0:  2d 7a 01 3e ff f0 2d 7a  01 3c ff f4 41 ee ff f0   -z.>.p-z  .<.tAn.p
00d0:  4e b9 00 00 26 44 62 00  01 24 20 2e ff fc 4e b9   N9..&Db.  .$ ..|N9
00e0:  00 00 2a fc 41 fa 01 22  4e b9 00 00 26 c4 4e b9   ..*|Az."  N9..&DN9
00f0:  00 00 22 38 2d 40 ff f8  20 2e ff f8 4e b9 00 00   .."8-@.x   ..xN9..
```

Now Newco's engineers must take these first few bytes and manually "disassemble" them, converting back from their hexadecimal notation into something more meaningful that will tell them what each instruction is telling the CPU to do. Disassembly is so called because it takes a small step backwards towards assembly language.

Life would be easy if all instructions on this particular CPU (a Motorola 68000 as Newco's engineers determined by visual examination of the chips) took the same number of bytes. The engineers could skip ahead and see what hex values caught their eye. However, the more complex instructions for the Motorola 68000 take more bytes. The only way to proceed is to disassemble each instruction from the first. As they disassemble the instructions, Newco's engineers not only decode what each instruction is, but how many bytes it occupies. This in turn tells them where the next instruction should start.

In order to disassemble the first instruction, Newco's engineers examine the bytes "4e 56 00 00 df." A Newco engineer can thumb through Motorola's 68000 Reference Manual to find a quick reference chart showing each of the instructions in numerical order. The engineer will find an entry for instructions that have the hex digit 4 in their first nybble. There are quite a few of these, so the engineer must narrow the search down by looking only for instructions that have 4e as their first byte.

The only such instruction is the LINK instruction, and the manual also indicates that the next nybble must be a 5 (which it is) and that the remaining nybble specifies the register to be used by the LINK instruction. In this case it is a 6. The manual also indicates that the two bytes of hex 00 and 00 are part of the instruction. The Newco engineers now know that the first instruction is a LINK with register 6 and an operand of 0000. This also tells them that the next instruction must start after the second byte of 00. The bytes following are "df fc ff ff ff," so the "df" must be the first byte of the next instruction.

Looking up "df" in their Motorola Reference Manual, the Newco engineers race toward decoding their second instruction. It is an ADD

instruction, and the subsequent bytes direct the CPU to add a negative value of 10 hex to the contents of register 7.[33]

Manual disassembly is extremely tedious. It takes about a minute to decode each instruction and to double check the results. What Newco's engineers are retrieving from the object code are the individual instructions generated by a C compiler running on a Sun workstation. All that Newco's engineers have established is that the first two instructions, if they were to write them in assembler, would be:

```
linkw a6,#0
    #-0x10,a7
```

Note that there are no comments and no symbolic variable names to guide Newco's engineers. All they have are the raw, low-level instructions that will be executed by the CPU. After five minutes work, they have only decoded two instructions. A modern program may consist of at least 300,000 such instructions. Assuming the engineers would take only 30 seconds to decode an instruction, this means that Newco's engineers would take 2,500 hours to complete the disassembly, and that would only tell them what the raw instructions were. They would still have no high level understanding of the code itself. Nevertheless, ten months (2,500 hours) later they would have a disassembled listing.

Why indulge in this mental self-flagellation? Newco's engineers could write another program to automatically disassemble the binary code and translate it back into assembly language instructions. This "disassembler" could translate the entire program back into assembly language in just a few moments. As it happens, every computer manufacturer since the early 1950s, including Sun Microsystems has provided such a program with the basic software that accompanies the computer.

When Newco's engineers run the disassembler program on the object code the first few lines of assembly language code would appear as:

33. For reasons that are, as yet, a complete mystery to Newco's engineers, disassembly reveals what a program does, not why it does it.

```
0000: orb    #0x10b,d2
0004: orb    #0x4000,d0
0008: orb    #0000,d0
000c: orb    #0xefc,d0

[ Lines deleted for brevity ]

0080: word     0x436f     invalid instruction
0082: moveq  #0x79,d0
0084: moveq  #00,d1
0086: movw     0x38350053:l,a0@-
008c: moveq  #0x6e,d2
008e: movl   a5,a0
0090: bvss   00f5
0092: moveq  #0x6f,d1
0094: orb    #0x4e75,d0
0098: jmp    0x2b90:l
009e: orb    #0x4e56,d0
00a2: orb    #0xffffdffc,d0
00a6: word   0xffff       invalid coprocessor instruction
00a8: word   0xfff0       invalid coprocessor instruction
00aa: moveml #0,sp@
```

Newco's engineers would quickly realize they have a problem. They started the disassembler at memory location 0000H and it tried to make sense of every instruction that it saw, but it clearly got "out of step," and started reporting invalid instructions. Even the two instructions the engineers know are at location 00a0 and 00a4 are not shown. The disassembler thinks that an instruction starts at 00a2, right in the middle of the first instruction that they manually disassembled. Clearly they need to repeat this process, this time telling the disassembler where the first instruction is.

This illustrates a severe weakness in using disassembler programs: they can only disassemble instructions if they are told where the instructions start. On many computers, code and data can be intermixed; a small block of code will be followed by data, then more code and more data and so on. Newco's engineers have a major problem before they can even run the disassembler: they have to know where all the instructions are. The problem is, as they have seen, that without disassembling the instructions, they do not know where the instructions are! Using a disassembler is a very time consuming and repetitive process. Newco's engineers must try disassembling some part of the program, inspect their results, and, using instinct as much as logic, adjust where the disassembler starts disassembling instructions and where it skips over data.

Nevertheless, with some coaxing, Newco's engineers can ultimately persuade the disassembler to produce the following listing (only the first few instructions are shown):

```
00a0: linkw  a6,#0
00a4: addl   #-0x10,a7
00aa: moveml #0,sp@
00ae: movl   #0,a6@(-4)
00b6: movl   a6@(-4),d0
00ba: jsr    0x0afc:l
00c0: movl   0x0200,a6@(-0x10)
00c6: movl   0x0204,a6@(-0xc)
00cc: lea    a6@(-0x10),a0
00d0: jsr    0x0644:l
00d6: bhi    01fc
00da: movl   a6@(-4),d0
00de: jsr    0x0afc:l
00e4: lea    0x0208,a0
00e8: jsr    0x06c4:l
00ee: jsr    0x0238:l
00f4: movl   d0,a6@(-8)
```

Again, it must be emphasized that no new information has been added. Only the representation of the binary patterns as they are in memory has been changed. There is an absolute, one-to-one relationship between the instructions that the disassembler outputs and these bit patterns in memory.

Furthermore, the above example illustrates one of the biggest single obstacles to reverse engineering: there is absolutely no high level of abstraction information present in the output from a disassembler. The instructions and their operands appear in stark detail, but Newco's engineers have absolutely no clues as to: (a) what these instructions are actually doing; (b) why they are doing what they are doing; and (c) when, in the overall program's execution, these instructions might be executed.

To the uninitiated eye, it appears that the instructions shown above are executed sequentially, starting from the first and proceeding down through each subsequent instruction. But this could be a completely fallacious inference as a brief examination by Newco's programmers could reveal. Here are the first few instructions again:

```
00a0: linkw  a6,#0
00a4: addl   #-0x10,a7
00aa: moveml #0,sp@
00ae: movl   #0,a6@(-4)
00b6: movl   a6@(-4),d0
00ba: jsr    0x0afc:l <<--This is a jump out of sequence to different code
00c0: movl   0x0200,a6@(-0x10)
```

Note the "jsr" instruction at location 00ba. This instruction tells the computer not to execute the following instruction at 00c0, but to "jump" out of sequence to a completely different part of the program. In this case, the program would jump to a part that Newco's engineers have not yet disassembled and therefore the purpose would be unknown.[34]

In the above example, Newco's engineers could discover that the computer does not execute the instruction at 00c0 until the program is about to terminate. The entire functionality of the program may be found at the destination of the "jsr" instruction, and the "movl" at 00c0 is part of the program's shut-down sequence. As Newco's engineers would be quick to tell us, this kind of convoluted, jumping backwards and forwards is quite normal. This example shows six instructions that will cause a discontinuity in the sequence of top-to-bottom execution of the instructions. It is this struggle to follow and comprehend the instructions that will challenge Newco's engineers every step of the way as they start to "decompile" the code.

e. Starting the "Decompilation" Process

The non-technical definition of "decompilation" describes the process as disassembling object code and attempting to recreate the original source code from the object code. Newco's engineers must now try to make sense of these instructions. As discussed above, they can see *what* the computer is doing, but they have no idea of the higher-levels of abstraction: Why is the code written the way it is? What is the processing sequence?

The first step in making sense of these instructions is for Newco's engineers to add their own comments to the disassembly listing. What they are trying to do, in their own faltering way, is to guess at what the program might be doing.

The example below shows, to the right of the instructions, some of the comments that the engineers might add:

---

34. It is as though Newco's engineers are on a conceptual treasure hunt at the house of a friend, following clues that lead them from the entrance hall, to the dining room, then the kitchen, only to then discover a clue that says, "Board a jet to Kathmandu and look in the largest hotel's lobby behind the third potted plant for your next clue." What is just one small clue in the treasure hunt could well turn out to be a life's work to follow.

```
00a0: linkw  a6,#0       Set up link word (not sure why?)
00a4: addl   #-0x10,a7   Add -10 hex to register A7 (why?)
00aa: moveml #0,sp@      Initialize program ready to run
00ae: movl   #0,a6@(-4)  Initialize link word (why?)
00b6: movl   a6@(-4),d0  Set up reg. D0 from A4 (what value?)
00ba: jsr    0x0afc:l    Call a subroutine (to do what?)
00c0: movl   0x0200,a6@(-0x10)      More setup stuff?
00c6: movl   0x0204,a6@(-0xc)
00cc: lea    a6@(-0x10),a0    Point A0 to A6-10H
00d0: jsr    0x0644:l    Call subroutine (to do what?)
00d6: bhi    01fc        Branch if "hi" condition on return (signifying what?)
00da: movl   a6@(-4),d0  Point D0 to A6-4 (why?)
00de: jsr    0x0afc:l    Call subroutine (see address 00baH)
00e4: lea    0x0208,a0   Load address 0208 into register A0
00e8: jsr    0x06c4:l    Call subroutine (to do what?)
00ee: jsr    0x0238:l    Call subroutine (to do what?)
00f4: movl   d0,a6@(-8)  Set A6-8 from D0
00f8: movl   a6@(-8),d0  Set D0 from A6-8 (this is redundant?)
00fc: jsr    0x0afc:l    Call subroutine (to do what?)
```

The example shows how little Newco's engineers know about the object code. They can see operations being performed, but have no idea why. There is a subroutine[35] located at location 0afcH that, if the number of times it is used is any indication, is apparently important. There are also other subroutines whose presence is shown by "jsr" instructions. "Jsr" means "jump subroutine"[36] and is used to direct the processor to break sequence, follow the instructions contained in the specified subroutine and then return to execute the instruction following the "jsr."

To understand what a particular subroutine does, Newco's engineers must painstakingly examine the disassembled output for the subroutine. If the subroutine itself contains "jsr" instructions that transfer control to other subroutines, then these subroutines must be disassembled and comprehended. This kind of "nesting"[37] is absolutely normal. Such nesting may occur to twenty or thirty levels in modern object code, especially object code generated by an "optimizing" compiler that translates the original source code, optimizes the object for execution, and then "links" it with prefabricated libraries of fine-tuned ob-

35. A subroutine is a small self-contained group of instructions used to perform a specific function. The computer stops executing the main program, starts executing the subroutine code, and when the subroutine is complete, returns to the main code. The act of executing the subroutine usually changes the contents of the CPU registers. The main program places the data to be processed by the subroutine into registers before transferring control into the subroutine. The subroutine also places return values in the CPU registers, overwriting their previous contents.

36. A jump subroutine is like a footnote. The reader stops reading the main text body, ducks down to read some additional information, and having read it, returns to the text following the footnote.

37. The "nesting" being referred to is subroutines calling subroutines calling subroutines.

ject code subroutines prepared by specially skilled systems programmers.

Any literary work that has twenty to thirty levels of indirection, referring the reader from paragraph 102 to paragraph 239, then to paragraph 97 and then to paragraph 3 and so on, would be deemed to be totally incomprehensible. Yet, this is completely normal in software. The computer itself does not "understand" the object code containing the instructions it is executing; it blindly executes one instruction after another and it does not matter to the computer that the instructions are scattered around the program rather than being physically contiguous. The poor Newco engineer must patiently plod through all of the subroutines calling other subroutines, struggling to comprehend what each subroutine does, and why it calls the other subroutines that call the other subroutines.

Only by continuing this process can Newco's engineers gain enough knowledge about the program and each of the subroutines it calls to make one or two tentative inferences as to what the program might be doing. Essentially, they are synthesizing a mental model of what the program does, using as ingredients: (1) the actual instructions that they see being given to the computer; (2) their skills as programmers to understand the significance of those instructions; and (3) their prior experience to infer the larger purpose of the code and the problem that it is trying to solve.

Newco's engineers still do not have any symbolic names to add any clues to this puzzle. The data variables and subroutines are devoid of any semantic information that might provide a hint of what is happening. After considerable study, the engineers can make intelligent guesses as to what some of these subroutines are doing by examining the object code. They could perhaps update their disassembly listing with some symbolic names to help them remember what the code was doing. In the example below, such names are shown in bold italics:

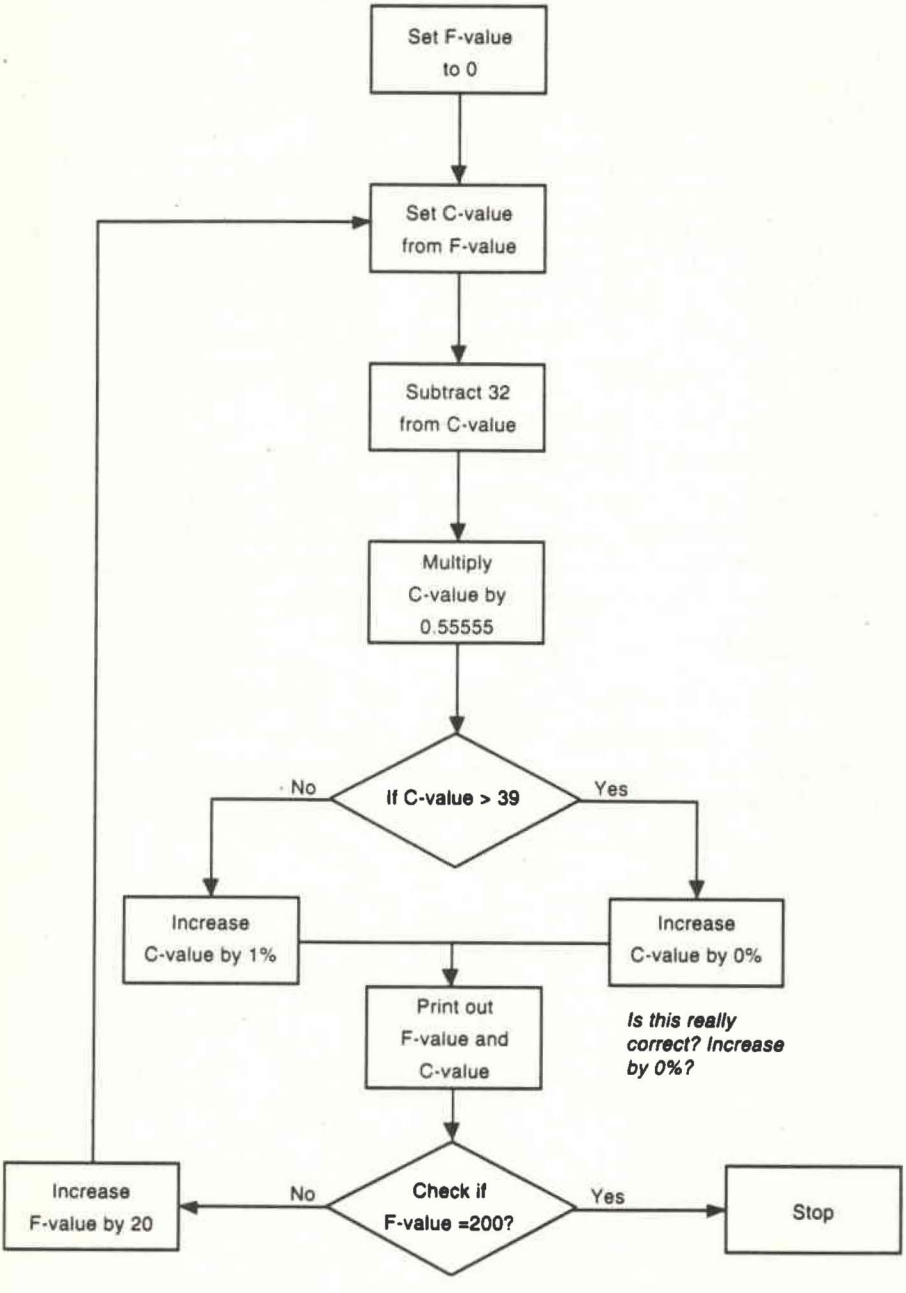
```
00a0: linkw  a6,#0       Set up link word (not sure why?)
00a4: addl   #-0x10,a7   Add -10 hex to register A7 (why?)
00aa: moveml #0,sp@      Initialize program ready to run
00ae: movl   #0,a6@(-4)  Initialize link word (why?)
00b6: movl   a6@(-4),d0  Set up reg. D0 from A4 (what value?)
00ba: jsr    Fstod       Convert single-length floating point number to double length.
00c0: movl   0x0200,a6@(-0x10)      More setup stuff?
00c6: movl   0x0204,a6@(-0xc)
00cc: lea    a6@(-0x10),a0    Point A0 to A6-10H
00d0: jsr    Fcmpd       Double-length floating point compare
00d6: bhi    01fc        Branch if "hi" condition on return (signifying what?)
00da: movl   a6@(-4),d0  Point D0 to A6-4 (why?)
00de: jsr    Fstod       Convert single-length floating point number to double length.
00e4: lea    0x0208,a0   Load address 0208 into register A0
00e8: jsr    Fsubd       Subtract double-length floating point
00ee: jsr    Fdtos       Convert double-length floating point to single-length.
00f4: movl   d0,a6@(-8)  Set A6-8 from D0
00f8: movl   a6@(-8),d0  Set D0 from A6-8 (this is redundant?)
```

Newco's engineers can gradually add their understanding of what the code appears to be doing instruction by instruction, subroutine by subroutine, symbolic label by symbolic label. The emphasis is that they are adding their ideas; the code does not contain any of the symbolic names, or the higher-level material that they are adding.

f. Creating A Flow-Chart

Finally, after many long weeks of research, disassembly, commenting, guessing, and perhaps some actual observation of the program running under the control of a diagnostic program, Newco's engineers can make the leap to a higher level of abstraction, the flow-chart:

Figure 2


Set F-value to 0
Set C-value from F-value
Subtract 32 from C-value
Multiply C-value by 0.55555
If C-value > 39
No
Yes
Increase C-value by 1%
Increase C-value by 0%
Print out F-value and C-value
Is this really correct? Increase by 0%?
Check if F-value =200?
No
Yes
Increase F-value by 20
Stop

For the first time, Newco's engineers can now see the overall "shape" of Sensatemp's algorithm. The symbolic names, F-value and C-value, were created by Newco's engineers to represent the two data variables for Fahrenheit and Celsius temperatures respectively. The conversion is performed by multiplying the F-value, less 32, by 0.55555 (the fraction 5/9 as a decimal number). Newco's engineers were puzzled to learn that if the C-value is greater than 39, the object code dutifully attempts to increase the C-value by zero percent! This explains why their observations of the Sensatemp output data showed that 200°F was exactly equal to the converted value without any apparent compensation. Can this really be true? Why write a program to calculate an increment of 0%? Why not just leave the value exactly as it was without any attempt to compensate? That remains a mystery.

g. Creating Source Code

For the purposes of this hypothetical example, a colleague of the author was given the flowchart and asked to play the role of the Newco programmer. He created the following source code:[38]

38. The characters enclosed in /* . . . */ are comments and play no part in the calculation. The reader should not attempt to make sense of the details of the program. The intent is to provide a general idea of the "shape" of the program.

```
/* ************************************************************
This program was written from a flow-chart provided.
It prints out compensated temperature values in Fahrenheit and
Celsius for values of Fahrenheit from 0 to 200 degrees using the logic shown
in the flow chart.
For temperatures less than 40°C, the temperature in °C is increased by one
percent.
************************************************************ */
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
      float F, C;
      F = 0.0;          /* Initialize Fahrenheit temperature to 0 */
      while(1)          /* Enter permanent loop */
      {
            C = F - 32; /* Subtract off 32 as start of conversion */
            C *=C * 0.55555;

            /* Check if compensation required (below 40°C)
            comp = 0.0;       /* Assume no compensation required */
            if (C < 40)       /* Apply compensation if C less than 40° */
                  Comp = 0.01;
                  C = C + (C * Comp);     /* Apply compensation */
            printf("\nFahrenheit = %f, Celsius = %f", F, C);
            if (F == 200)           /* If F gets to 200, we are finished */
                  break;            /* Break out of while loop */
            else
                  F += 20           /* Increase F temp by 20 */
      }
}
```

h. The Original Source Code

Did the Newco programmer recreate the "original" source code? Only a comparison of the Newco source code to the Sensatemp code (written by this author) will show the answer to the question. The Sensatemp source code is as follows:

```
/*
 * ftoc.c
 *
 * Copyright (c) 1992, Sensatemp Inc.
 * This source code contains proprietary information and trade
 * secrets of Sensatemp Inc. and may not be reproduced in any
 * form without the written permission of Sensatemp. Inc.
 * This program contains a top-secret formula for converting
 * temperatures in fahrenheit to centigrade for the purpose
 * of displaying the results of the XYZ Inc. temperature probe.
 * Because of a non-linear response from the XYZ Inc's probe,
 * the need exists to increase the final temperature by 1% per
 * cent
 * for temperatures less than 40 C, and by 5% for temperatures
 * above that.
 * The regular formula is to subtract 32 and to multiply by
 * 5/9.
 *
 */

#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{
      float DegF, DegC;
      for (DegF = 0.; DegF <= 200.; DegF += 20.)
      {
            DegC = DegF - 32.;      /* Subtract off the 32 */
            DegC *= 0.55555555;     /* 5/9 */
            if (DegC <= 39.)        /* Check which correction apply */
            {
                  DegC = DegC + (DegC * 0.01); /* 1 percent */
            } else
            {
                  DegC = DegC + (DegC * 0.0); /* 5 percent */
            }
            printf ("Temp. F = %7.4f, C = %7.4f\n", DegF, DegC);
      }
}
```

To appreciate the differences and similarities between the source code produced by reverse engineering and that originally written by Sensatemp, small fragments of the two pieces of source code must be compared.

The Newco code reads:

```
/* ************************************************************
This program was written from a flow-chart provided.
It prints out compensated temperature values in Fahrenheit and
Celsius for values of Fahrenheit from 0 to 200 degrees using the logic shown
in the flow chart.  For temperatures less than 40°C, the temperature in °C is
increased by one percent.
************************************************************ */
```

The corresponding part of the Sensatemp code reads:

```
/*
 * ftoc.c
 *
 * Copyright (c) 1992, Sensatemp Inc.
 * This source code contains proprietary information and trade
 * secrets of Sensatemp Inc. and may not be reproduced in any
 * form without the written permission of Sensatemp. Inc.
 * This program contains a top-secret formula for converting
 * temperatures in fahrenheit to centigrade for the purpose
 * of displaying the results of the XYZ Inc. temperature probe.
 * Because of a non-linear response from the XYZ Inc's probe,
 * the need exists to increase the final temperature by 1% per
 * cent
 * for temperatures less than 40 C, and by 5% for temperatures
 * above that.
 * The regular formula is to subtract 32 and to multiply by
 * 5/9.
 *
 */
```

Newco's initial comment block is quite different from Sensatemp's, both in terms of what it says, and in the formatting. Newco used a horizontal line of asterisks above and below the initial comment block to highlight it. Sensatemp has used a different style, with a line of asterisks down the left hand edge of each line to create a sort of change bar effect.

The Sensatemp comments reveal that there are two apparent mistakes in Newco's code. First, Sensatemp claims to compensate differently for temperatures less than *or equal to* 40°C and those above 40°C. Newco's code applies compensation for temperatures *less than* 39°C. It appears that Newco's engineers are in error; they should have checked for temperatures *less than* 41°C to follow Sensatemp's comment. Furthermore, Sensatemp's comment says that *above* temperatures of 40°C, Newco should be compensating by a factor of 5%! For temperatures above 39°C, Newco should not apply any compensation.

These mysteries can be resolved by further examination of the source code. Newco's code reads:

```
********************************************************** */
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
```

The corresponding fragment of Sensatemp's code reads:

```
*/

#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{
```

Newco's code looks identical to Sensatemp's! Is this clear evidence of slavish copying? The short answer is "No!" All of these lines are absolutely standard for any program written in the C language and therefore the similarity, such as it is, is a direct product of the constraints under which the programmers developed the program.

There are small signals that Newco's code is not a slavish copy of Sensatemp's. First, Newco's code has no blank line after the comment block before the standard statement "#include <stdio.h>". Second, Sensatemp's programmer wrote "main(argc,argv)" without a space after the comma. Both programs are equally correct. They are similar but not infringing.

Moving to the next code fragment, here is the Newco code:

```
float F, C;
F = 0.0;          /* Initialize Fahrenheit temperature to 0 */
while(1)          /* Enter permanent loop */
{
      C = F - 32; /* Subtract off 32 as start of conversion */
      C *=C * 0.55555;

      /* Check if compensation required (below 40°C)
      comp = 0.0;             /* Assume no compensation required */
      if (C < 40)             /* Apply compensation if C less than 40°
*/
            Comp = 0.01;
      C = C + (C * Comp);           /* Apply compensation */
      printf("\nFahrenheit = %f, Celsius = %f", F, C);
      if (F == 200)                 /* If F gets to 200, we are finished
*/
            break;                  /* Break out of while loop */
      else
            F += 20;                /* Increase F temp by 20 */
}
```

And here is the corresponding Sensatemp code:

```
float DegF, DegC;
for (DegF = 0.; DegF <= 200.; DegF += 20.)
{
      DegC = DegF - 32.;      /* Subtract off the 32 */
      DegC *= 0.55555555;     /* 5/9 */
      if (DegC <= 39.)  /* Check which correction apply */
      {
            DegC = DegC + (DegC * 0.01); /* 1 percent */
      } else
      {
           DegC = DegC + (DegC * 0.0); /* 5 percent */
      }
      printf("Temp. F = %7.4f, C = %7.4f\n", DegF, DegC);
}
```

This is the heart of the program. It shows declarations of symbolic variable names, the logic that makes the program loop around several times, and the calculations to convert temperatures and print them out. Newco's code uses the names "F" and "C" for the two data variables to contain the Fahrenheit and Celsius temperatures respectively. Sensatemp's code uses "DegF" and "DegC" respectively. The codes are similar, of course, but not identical.

To achieve the effect of looping around several times, Newco's code uses what is called a "while" loop (i.e. while a condition is true, execute the next block of code—the intent being that the code in the block will somehow change the condition so that the computer ultimately comes out of the loop). In fact, Newco's code uses a conventional means of making the loop appear infinite by saying "while(1)." The only way out of this kind of loop is a "break" statement at the end of Newco's "while" loop. The "break" statement is only executed if the temperature in F reaches a value of 200. If the temperature in F is not

equal to 200, then the code increases the temperature in F by 20 and returns to the top of the "while" loop.

On the other hand, Sensatemp's code uses a completely different construct, a "for" loop, that initializes DegF to 0, and each time around the loop checks to see if DegF is less than or equal to 200. If it is less than or equal to 200, the computer executes the loop one more time having incremented DegF by 20. Clearly Newco's logic for looping around is very different. But how could this be? Newco's engineers disassembled the actual object code produced by the compiler for Sensatemp's source code. Why does Newco's source code not conform to Sensatemp's?

The answer is that the compiler Sensatemp used deliberately generated object code that was a simpler version of the "for" loop. Newco's engineers were guided by the compiler output (the object code) not the Sensatemp source code.

Two more mysteries are also revealed. Regardless of what Sensatemp's documentation said the program should do, it applies 1% compensation for Celsius temperatures *less than or equal to 39°C.* Furthermore, examination of the source code line that does compensation for temperatures above 39°C shows the following:

```
DegC = DegC    (DegC * 0.0); /* 5 percent */
```

The comment says 5%, but the source code on the left says 0%. The fact that this comment and the one in the header block both say 5% indicates that Newco engineers have faithfully reverse engineered a mistake in Sensatemp's program. Such a copied mistake has, in other circumstances, been used as "proof" of copyright infringement.

III. Truths of Reverse Engineering

The preceding example has shown the realities of reverse engineering as a process of painstakingly attempting to understand the ideas embodied in the object code of a computer program. Revealing the truths of reverse engineering in practice also demands that previously disseminated falsehoods be shown for what they are. Therefore, this paper concludes by addressing some of these technical falsehoods revealed either in articles opposing reverse engineering, or by the misunderstandings of some courts when confronted by the very confusing and apparently slippery concepts associated with computer science.

A. *Is Software Reverse Engineering Necessary?*

The opponents of reverse engineering who filed an Amici Curiae Brief[39] in *Sega Enterprises Ltd. v. Accolade, Inc.* stated:

> The argument that it is necessary to copy and adapt the object code version of a copyrighted computer program to understand its "ideas" is without merit. Alternative means are available to study a program and analyze how it operates. For example, a developer who wishes to learn about a program can: read the documentation, user manuals and other materials published by the developer; observe screen displays; observe the program in operation, studying input, output, and the speed with which the program functions are performed; read and study the object code; perform timing tests; test the programs' functions by designing input data specifically for that purpose; and, by attaching test equipment, physically examine the internal parts of the computer while the program is running.[40]

This argument is technically naive as can be seen if each suggested alternative is considered in a real world context.

The argument assumes several points. First, it assumes that documentation is available. No relevant documentation, however, is available from Nintendo or Sega. Second, it assumes that the documentation contains all of the requisite information. In reality, in most instances where reverse engineering is done, it is only done precisely because the documentation, if any, fails to provide the required information. Third, it assumes that the program even produces screen displays; but, the essential code in both the Sega and Nintendo units operates invisibly. It also assumes that the input and output are not encrypted and are comprehensible. The Nintendo base unit outputs long streams of pseudo-random 0s and 1s with pauses of pseudo-randomly determined length. It further assumes that timing computer software gives significant clues to its function. The execution time of all but the most specialized software is more affected by the sophistication of the compiler than the underlying algorithm.

The argument makes further assumptions. For instance, it assumes that modern object code is as simple and as small as software was back in the dawn of computing. As this paper has demonstrated, however, a modern program is hundreds of times larger than those early programs,

---

39. The amici were IBM, Apple Computers, Autodesk, Computer Associates, Digital Equipment Corporation, Intel Corporation, Lotus Development Corporation, WordPerfect Corporation and Xerox Corporation.

40. Amici Brief, *supra* note 12, at 18 (footnotes omitted). "It is possible to read object code. . . . Indeed prior to the advent of assemblers and compilers in the early 50's, all programming was done in machine language." *Id.* n.26 (citations omitted).

and the binary representation of each instruction is sufficiently arcane that, taken together, human beings cannot, in a real and practical sense, understand object code without making some kind of intermediary copy. Next, it assumes that testing the programs' function by contriving special input data is feasible or likely to yield relevant results. But, the relevant code in *Sega* and *Atari* consisted of a "private" conversation between two central processing chips. There was no means of creating "special input data." The argument further assumes that the test equipment attached to examine the internal parts of the computer can operate without making any kind of intermediary copy. Such devices almost always produce such a torrent of information that some kind of printout is required to understand all but the most minuscule fragment of object code. Furthermore, such devices almost always include disassemblers to convert the binary data into a human comprehensible form.

The Amici's assertions are bogus. The reality of the program only exists within the object code. It is the object code alone that can answer every question.

B. Decompilation Is Not Used For Developing Original Computer Programs

Later, in this same *Amici Curiae* Brief, the *Amici* state: "Decompilation is not standard industry practice in developing original computer programs."[41] This is untrue because it presumes that "original computer programs" are developed and run on a computer in isolation from all other programs. That was true in the early 1960s when the big mainframe computer was king. Today, however, nothing could be further from the truth. As this author writes this paper on a laptop computer using Microsoft Word for Windows, there are eight major software products that must interoperate flawlessly, executing billions of computer instructions as the computer darts backwards and forwards between each of the eight products.[42] Imagine the company that develops just one of these eight software products. What will it do if it discovers an unexpected system crash during the development of its software? If research shows that the problem does not lie in its own software, it would have no option but to try and discover what peculiar interaction between its software and the other seven programs would cause the crash. It would immediately start to reverse engineer one or

41. *Id.* at 29.

42. Microsoft Windows, Microsoft Disk Operating System, Adobe Type Manager, Word for Windows, Alki's MasterWord extensions to Word for Windows, Norton's Desktop For Windows, Stac Electronics' Stacker software that doubles the hard disk space, and QEMM managing the computer memory.

all of the other programs, following its instincts to determine which program might be the culprit.

A recent excerpt from an industry trade magazine suggests that even IBM may have to use reverse engineering to maintain compatibility for its OS/2 operating system:

> As its long-standing agreement with erstwhile partner Microsoft Corp. draws to a close this week, IBM's Personal Software Products unit finds itself at a cross-roads. While its rights to Microsoft's 16-bit DOS and Windows code will ensure OS/2 sufficient compatibility for the bulk of installed applications for at least the next year or so, observers call into question IBM's ability to sustain future support in a timely fashion. . . . Most agree that while it is feasible to reverse-engineer the APIs[43], IBM will surely be in the undesirable position of playing catch-up if it wants to maintain compatibility with Microsoft's platforms.[44]

It would appear that even IBM may now find itself forced to reverse engineer in order to keep its OS/2 operating system capable of running programs originally designed to run under Microsoft Windows. Absent a license with Microsoft, and presuming that IBM wishes to run Microsoft Windows applications programs under OS/2, IBM will have to reverse engineer and disassemble parts of future versions of Windows. Its own counsel stated in a paper delivered at the 1993 University of Dayton Intellectual Property Symposium that: "[Disassembly] is without question an attempt to obtain information that the right-holder lawfully seeks to withhold from its competitors."[45]

C. As Programmers Can Read Object Code Directly, Is Disassembly/Decompilation Necessary for Reverse Engineering?

Opponents of decompilation and reverse engineering make their argument sound reasonable.[46] Therefore, they are able to agree, and they do so with amazing consistency. Computer programmers, in the experience of this author, inevitably greet the opponent's assertion with derision and disbelief. In consideration of its proponents, the argument must be taken seriously, at least long enough to show the bogus technical foundations on which it rests. These foundations are that: programmers can read *and comprehend* object code directly when viewed on a computer screen *without the need to make notes*; and the act of viewing object code directly obviates the need for any infringing intermedi-

---

43. Knowledge of Application Program Interfaces is necessary to maintain compatibility and allow programs designed to run under Windows to run under OS/2.

44. Amy Cortese, *IBM Faces Hurdles with OS Strategy*, PC WEEK, Sept. 13, 1993, at 57, 68.

45. Clapes, *supra* note 32.

46. Clapes, *supra* note 32.

ary copies. Both of these arguments are built on technically faulty foundations. On closer examination, they are bogus, especially when judged by the metrics of infringement advanced by these very opponents of disassembly and "decompilation."

While it is true that some of the more skilled programmers of the world can read object code directly (including this author for certain Intel CPU chips), the real issue is not one of perception but of comprehension. Modern programs, as even the simplistic Sensatemp hypothetical example showed, are not executed instruction-by-instruction from the first to the last. The actual execution flow darts backwards and forwards like a demented waiter taking orders from diners at widely separated tables in a restaurant. One subroutine calls six other subroutines; each of these six may call another six subroutines and each of these thirty-six subroutines call yet more subroutines. A skilled programmer, attempting to comprehend the program (as opposed to merely trying to read the object code), must grasp many different aspects of the program mentally. Many of these aspects are only dimly understood.

The challenge facing programmers trying to understand even the most basic program is to hold all of this information in their brains until they can resolve the numerous mysteries and unanswered questions. Even as early as 1956, cognitive psychologists such as Professor George Miller in his now classic paper *The Magical Number Seven Plus or Minus Two*[47] demonstrated how painfully small a human's "short term memory" really is. Such memory is capable of holding only seven (plus or minus two) cognitive "chunks" of information at any moment in time. This limited "working memory" (as it came to be called) is shown, with graphic clarity, by Professor Ben Shneiderman in his book "Software Psychology."[48] Opponents of reverse engineering should read this book to really understand why programmers cannot read and comprehend large quantities of object code, or even source code.

Speaking specifically of how the fact that programs jump backwards and forwards inhibits comprehension (and bear in mind this was source code resplendent with massive amounts of high-level of abstraction information), Professor Shneiderman says: "[F]orward or backward jumps would inhibit 'chunking' (the name given to the process of mentally gluing quanta of information to form higher level concepts)

47. George A. Miller, *The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information*, 63 THE PSYCHOL. REV. 81 (1956).

48. BEN SHNEIDERMAN, SOFTWARE PSYCHOLOGY: HUMAN FACTORS IN COMPUTER AND INFORMATION SYSTEMS 46-54 (1980).

since it would be difficult to form separate chunks without shifting attention to various parts of the program."[49] The impossible situation confronting a programmer trying to read and comprehend object code viewed through the keyhole of a computer display screen could be ameliorated if the programmer were to take written notes, jotting down quanta of information that could not yet be "chunked" into concepts.

This very idea was put forward by an opponent of decompilation and disassembly[50] as a "work around" for a programmer's inability to comprehend large quantities of convoluted object code. It is a curious and contradictory idea which falls prey to the additional argument of those who oppose decompilation and disassembly (and echoed by Sega's attorneys) that any such notes would be a derivative work, containing, as they would, detailed expressive information copied directly from the original object code.

The notion that, by confining themselves to reading object code directly on the computer's display screen, programmers obviate the need to make intermediary (and infringing, as these opponents would argue) copies of the object code, is similarly curious, contradictory and ill-informed. According to the Copyright Act of 1976:

> 'Copies' are material objects, other than phonorecords, in which a work is fixed by any method now known or later developed, and from which the work can be perceived, reproduced, or otherwise communicated, either directly *or with the aid of a machine or device*. . . . A work is 'fixed' in a tangible medium of expression when its embodiment in a copy or phonorecord, by or under the authority of the author, is *sufficiently permanent or stable to permit it to be perceived, reproduced, or otherwise communicated* for a period of more than transitory duration.[51]

Any intermediary copy of the object code made in order for a programmer to read and comprehend it would, of necessity and definition, meet this fixation requirement. It requires considerably more than "transitory duration" for even the most skilled programmers to stare at the computer display screen and divine what the program is doing. It is technically contradictory to argue that programmers can perceive the code (and take the time to comprehend it), and yet, such copies would not be "fixed" because they are transitory. If the copies were transitory, then the programmers could hardly comprehend them!

The second fundamental technical flaw in the foundational argument of those who oppose disassembly and decompilation is that viewing object code on the computer display screen obviates the production

---

49. *Id.* at 53.
50. *See generally* Clapes, *supra* note 32.
51. 17 U.S.C. § 101 (1988) (emphasis added).

of intermediary copies of this object code. Not only is this false, but it can be shown by even a novice programmer, that the display of object code on the computer screen results in the same number of intermediary copies being created as if a programmer were to view disassembled source code on a computer screen. To illustrate this point demands a brief excursion into the inner world of the computer. Imagine a program whose purpose is to read in object code and display it on the computer's display screen, presenting screenful after screenful, and allowing a programmer to page backwards and forwards within the object code. For subsequent clarity, let this program be called HEXDUMP.

Consider the inner operations that must occur within an IBM personal computer when the HEXDUMP program displays the first screenful of hexadecimal object code of, say, the main Lotus 1-2-3 program for use under MS-DOS (which, incidentally, requires over 900,000 bytes of memory). The following sets forth the specific operations that will occur before HEXDUMP can display the first few bytes of the program on the screen. First, HEXDUMP makes a request to the MS-DOS operating system for the first 512 bytes of the 1-2-3 program to be read into a dedicated area of memory within the HEXDUMP program itself. Second, the operating system transmits a command to the hard disk drive controller to read the hard disk "sector" (usually 512 bytes in size) containing the first part of the 1-2-3 program. To do this, the hard disk controller, a computer in its own right, makes a copy of the required sector in a part of its own private memory (this is memory on the hard disk controller and not part of the main computer memory). Third, the hard disk controller then, in cooperation with the electronics in the main personal computer, makes a copy of the first 512 bytes of the 1-2-3 program in that part of the memory under control of the operating system. Fourth, the operating system, sensing that the requested data has arrived in that part of main memory that it has dedicated to storing hard disk information, then makes another copy of the first 512 bytes of the 1-2-3 program to the designated area of main memory within the HEXDUMP program itself. Fifth, the HEXDUMP program is then put back in control of the machine. It must take the pure binary information it finds in the memory area containing the object code from the hard disk and convert it into a form that can be displayed on the computer's screen. It accomplishes this by grouping the actual binary digits in memory, and substituting different binary values in another working area of memory that, when displayed, will reflect the true binary found in the object code. Recall the output of a typical hexadecimal dump: the left hand side shows the hexadecimal values of each byte, while the right side shows

these same bytes but viewed as ASCII. To perform this step, two more copies are made of the 1-2-3 object code. Finally, to display the converted binary code, HEXDUMP must request that the operating system copy the converted image into a special area of main memory, the so-called Video RAM (Random Access Memory), from which the hardware of the IBM personal computer will illuminate the phosphor dots on the computer screen so that the individual numbers and letters can be read by the programmer. Thus, four more copies are made—two in Video RAM (hexadecimal and ASCII) and two on the phosphor of the display screen (hexadecimal and ASCII). This process, offered as a means of avoiding intermediary copies, actually makes nine intermediary copies. Ironically, if the same programmer used a disassembler to display the first few bytes of the 1-2-3 program, that process too, would require nine intermediary copies, corresponding one for one, for the copies described above.

It will likely be argued that these copies are not copies under the meaning of the Copyright Act because they are not "fixed", but transitory. Is this really true? If Newco's valiant programmer stared at the first screenful of object code for fifteen minutes (assuming there was no other activity relating to the hard disk) all nine copies of the object code would remain completely intact. Are these transitory in nature? The court in *MAI Systems Corp v. Peak Computer, Inc.*[52] ruled that such RAM-based copies would be temporary, but not transitory:

> RAM can be simply defined as a computer component in which data and computer programs can be temporarily recorded. Thus, the purchaser of [software] desiring to utilitize the programs on the diskette could arrange to copy [the software] into RAM. This would only be *a temporary fixation.* It is a property of RAM that when the computer is turned off, the copy of the program recorded in memory is lost.[53]

A temporary fixation is still a fixation which raises two questions. First, what if the computer is not turned off, or the contents of RAM are preserved with a small backup battery power supply, as happens in most models of Toshiba laptop computers? Second, would that temporary fixation effectively become permanent if preserved for long enough? Opponents of disassembly and decompilation cannot have it both ways. Either the intermediary copies are fixed or they are not. The arbiter of fixation cannot be whether the screen is displaying hexadecimal object code or disassembled source code.

---

52. 991 F.2d 511 (9th Cir. 1993) (quoting Apple Computer, Inc. v. Formula Int'l, Inc., 562 F. Supp. 775 (C.D. Cal. 1983), *aff'd*, 725 F.2d 521 (9th Cir. 1984)).

53. *Id.* at 519 (emphasis added).

In the real world both the hexadecimal dump and the disassembled source code would be printed so that the programmer could comprehend it. The cognitive task of trying to memorize page after page of either is practically impossible. Such printouts are, to a programmer, clearly fixed within the definition of 17 U.S.C. § 101. It could also be argued that the intermediate copies produced by displaying hexadecimal object code on the computer's display screen are permissible under the terms of 17 U.S.C. § 101, on the basis that:

> [I]t is not an infringement for the owner of a computer program to make or authorize the making of another copy or adaptation of the computer program provided: 1) that such a new copy or adaptation is created as an essential step in the utilization of the computer program in conjunction with a machine and that it is used in no other manner. . . .[54]

This too, is technically fallacious. The 1-2-3 program's object code is being processed entirely as data, not as a computer program that will control the operations of the computer itself. It is therefore not "an essential step in the utilization of the computer program."[55] Those copies that are made to bring the object code from the hard disk into the main memory would be indistinguishable from those made to execute the program because at that stage the process is identical. But there the similarity ends; the object code never gets to control the computer but is merely processed by the HEXDUMP program as incoming data. As a software practitioner, the author also wonders whether, in this context, a reverse engineer is legally an "owner" or a "licensee."

D. Can Disassembly/Decompilation Be Used To Make A Program Run On A Different Computer?

It has also been argued that disassembly and decompilation can be used to migrate a program running on an Intel 80486 CPU chip to run on a Motorola 68040 CPU chip, for example. This implausible process is alleged to operate by extracting a program's assembly language source code from the Intel 80486 object code, translating that source code into Motorola 68040 assembly language source code, and passing the 68040 assembly language source code through an assembler program to form a new version of the binary object code that, *mirabile dictu*, will be ready to run on a Motorola 68040 machine.

There are several technical fallacies with this argument. The internal architecture of the Intel 80486 is very different from that of the Motorola 68040. The 80486 stores binary numbers in a radically differ-

---

54. 17 U.S.C. § 107 (1988).
55. *Id.*

ent way[56] than the 68040, so much so that assembly language code written for the 80486 will simply not work in the 68040 without major modifications to the overall logic. Similarly, the internal storage registers in the 80486 operate in a completely different way from those in the 68040, sufficiently different to confound any automated transliteration program. The differences in the two CPU chips and their respective operating methods further conspire to ensure that the very logic of the original 80486 program would not work if simply transported to the 68040. Assuming for a moment that the original 80486 program was designed to run on an IBM personal computer (PC) under Microsoft Windows, and the Motorola 68040 is destined to run on an Apple Macintosh computer running System 7.1 of the Macintosh operating system (and these are merely typical examples), there is absolutely no chance, even if the transliteration from one CPU to another could be done, that the program will work correctly. The hardware of the two computers is completely different, the operating systems of the two computers is completely different, and at higher levels of abstraction, the whole software environment of the two computers is completely different. In practice not even software developers "port" (as the act of migration is called) their software from the PC to the Macintosh; they rewrite it. If such developers, armed with all of the high level of abstraction information that they have created, cannot translate their own programs, what chance does someone armed with just a PC disassembler and Macintosh assembler have? They have absolutely no chance.

This suggested use of disassemblers is ill-founded. Its assertion is based on technical capabilities that simply do not exist. Even if such capabilities could be made to exist by sheer force of will, they fail miserably because of the differing software and hardware environments that prevail in different computer systems.

E. Does Reverse Engineering Lay Bare A Program's Inner Secrets?

Reverse engineering does not lay bare a program's inner secrets. Indeed, it *cannot*. The inner secrets of a program, the real crown jewels, are embodied in the higher levels of abstraction material such as the source code commentary and the specification. This material never survives the process of being converted to object code. As the inner secrets of a program are not in the object code, reverse engineering cannot lay them bare.

---

56. Large numbers require multiple adjacent bytes of binary digits, and are stored with the least significant byte first on the Intel 80486. On the other hand, the Motorola 68040, more in tune with the way humans store numbers, places the most significant binary digits first.

Furthermore, the implication is that reverse engineering is a process of distillation, removing information from the original object code to form an understanding of what the original source code must have been. As the preceding examples have shown, almost all of the information about the program (other than the low-level assembly language instructions) comes from the mind of the reverse engineer. In other words, reverse engineering is almost entirely an additive process, with the reverse engineer adding his or her knowledge and experience to the meager information contained within the object code.

F. What Information Does Reverse Engineering Reveal About A Program?

Reverse engineering can only reveal information contained within the binary object code being studied. In the event it appears to reveal more than that, it is actually information being supplied by the reverse engineer. In practical terms, and given considerable time, a skilled reverse engineer can divine precise, but partial, information about *what* a program does (as distinct from *why* it does it) in the following areas. First, information can be divined concerning the user interface. That is, the externally visible visual and audio interface perceived by someone using the program. It could be argued that reverse engineering this information is perverse as, in most cases, this information can be obtained more easily by Black Box observation of the program in operation.

Second, information can be divined concerning most of the internal interfaces[57] between the individual blocks of software being analyzed. There are inevitably some interfaces whose existence can be seen but, like some unused back country roads, never used. The reverse engineer, absent any information to the contrary, can only assume that if an interface exists, sooner or later someone will use it.

Third, information may be divined concerning most of the internal and external data structures used by the program under scrutiny. The existence and purpose of internal data structures can only be analyzed by observing the program placing data into, or retrieving data from, the various data fields within the structure. In many cases, the reverse engineer will observe that there are data fields within a data structure that never appear to be used no matter how diligently all of the different capabilities of the program are exercised. These data fields,

---

57. An interface is a software "connection" between one body of software and another. This author suggests a definition of an interface as "a point in an information processing system through which information passes without any intentional change to its format or its meaning." Opponents of reverse engineering deny that an adequate definition for an interface exists.

whether they are internal to the program or contained within a data file created or read by the program, will remain a mystery until they are seen in use. Data files can also be analyzed by creating special test data with predetermined data values in each data field. By analyzing the contents of these files, the reverse engineer can infer what information is being stored and how it is represented. This knowledge, however, is not all-seeing. Many times the reverse engineer will see a single predetermined data value that provokes numerous data fields to be set to special values. The reverse engineer can only speculate as to what these values represent unless he or she can reverse engineer a program that actually uses one or more of these values in the course of its operation.

Also, information can be divined concerning some aspects of the algorithms used by the program under scrutiny. Absent the high level of abstraction information, the reverse engineer can observe what the algorithms do, at least in those test situations that can be induced while the program is being observed. It is not at all unusual, however, that only ten to thirty percent of the object code is executed during normal operation of a modern program. The remaining seventy to ninety percent is reserved for error handling and special conditions that the reverse engineer might not be able to recreate. Although the amount of code executed will vary from one program to another, the real point is that it is unlikely that the reverse engineer will enjoy the luxury of observing all of the program's code in action. This fact makes discernment of specific algorithms much harder or even impossible.

Another piece of information which a reverse engineer can gather from the process of reverse engineering is the overall static structure of the object code, revealing how the object code is laid out when the program is loaded into memory to be run on the computer. This static structure of the object code is vastly different from the static structure of the source code. The source code has been translated by a Compiler program and glued together with other object code by a Linker program. The final object code is more a product of the Compiler and the Linker than the programmer who wrote the source code.

Reverse engineering also reveals information concerning the dynamic structure and execution sequence, the chronological sequence in which parts of the program are executed by the computer. This, however, will take the form of a vast amount of information, so large in fact that most reverse engineers would not find the information useful except where it can be focused on some detailed arcana that eludes understanding by other easier means.

G. What Can Never Be Discovered By Reverse Engineering?

No matter how talented the reverse engineer, and no matter how much time and money is dedicated to the task, software reverse engi-

neering can never recreate any of the following areas of information. First, it cannot reveal the original higher levels of abstraction information contained in design documentation, specifications, or business plans. The object code form of the program is devoid of this kind of information and the reverse engineer cannot therefore recreate it.

It also cannot reveal the original source code, complete with its commentary. This too, is simply not present in the object code. It cannot therefore be recreated. The original data structures, complete with data fields that might be set aside for future use, can also not be revealed. These will never be used by the program as it executes, and therefore, their purpose cannot be divined.

Reverse engineering cannot determine the original design rationale. The reverse engineer can discern what a program is doing, but not the underlying reasons why it does it the way it does, or why it does it one particular way rather than another.

H. Can Reverse Engineering Show Current and Future Compatibility Requirements?

Reverse engineering cannot show current and future compatibility requirements. The court in *Atari Games Corp. v. Nintendo of America, Inc.*[58] committed a major technological error when it stated:

> Program code that is strictly necessary to achieve current compatability [sic] presents a merger problem, almost by definition, and thus is excluded from the scope of any copyright. A defendant may not only make intermediate copies of an entire program to discover the existence of such code, but it may also copy the code into its final product. In contrast, program code that relates only to future compatability [sic] has no current function and thus cannot merge with the expression of any idea. Such code is therefore entitled to copyright protection.[59]

The court failed to understand that reverse engineering cannot tell whether a given feature is required for current or future compatibility; it can only show whether a given feature is in current use or not. Current use is determined by actual observation of the execution of the computer program under a variety of test circumstances. If the reverse engineer fails to see any use of the feature, he or she simply cannot know whether it is because the feature is a vestigial remnant of some previous requirement, or some embryonic future requirement for tomorrow or years hence.

---

58. 975 F.2d 832 (Fed. Cir. 1992).

59. Atari Games Corp. v. Nintendo of Am., Inc., No. C 88-4805 FMS, C 89-0027 FMS, 1993 U.S. Dist. LEXIS 6786, at *4 (N.D. Cal. May 18, 1993). "In other words, there is only one way to express the idea of generating a signal stream that unlocks the NES console." *Id.* n.3.

Atari's challenge was to obtain samples of all of the different types of Nintendo base units in use in the world today in order to establish overall compatibility requirments. It must be remembered that Nintendo markets different models for each country or major geographical region in the world. Only by observing all of the known Nintendo models in operation could Atari have determined whether or not a specific internal interface was in current use or not.

Given that Atari could have examined all of the known models, and could determine that none of these base units used the interface in question, it still would make absolutely no sense whatsoever for Atari, either by choice or court edict, to simply ignore an apparently unused compatibility requirement. If a compatibility requirement is seen to exist, one must assume that sometime, sooner or later, it will be brought into play. To fail to implement the same feature because one cannot see it being used is to commit economic suicide in the computer industry. Imagine distributing hundreds of thousands of game cartridges without this compatibility requirement only to wake up one morning and discover that new base units sold only in Europe now demand that the requirement be present. Worse yet, imagine that the base units are capable of maintaining the date with an internal clock chip and that on February 4, 1994, all base units, in unison, will use that compatibility requirement from then on.

I. Are Copied Bugs Always A Sign Of Infringement?

If an allegedly infringing program is found to contain some (or all) of the same mistakes as the original, it would be easy to conclude that the apparently infringing program had been slavishly copied from the original. Reverse engineering does not necessarily guard against this. If the original program has an error of logic, that error (if the reverse engineering is done properly) should appear in the subsequent code. If it does not, that would tend to point towards some skullduggery. How did the reverse engineers know how to correct this mistake? How did they know that it *was* a mistake?

J. Can Reverse Engineering Be Used To Disguise Copying?

Some opponents of software reverse engineering have asserted that the process provides the would-be software thief with a cheap method of eviscerating a successful software product. They claim that by shuffling the components into a new arrangement to disguise their origins, they can thereby reap unjustified enrichment on the intellectual coattails of the original program's owner.

Asserting that software reverse engineering allows people to shuffle a program's internal source code, thereby creating a disguised copy of the original, is both technical and commercial nonsense. If a thief

merely shuffles the original source code, the resulting object code will still contain numerous indicia of the software's origin and the theft would be readily exposed. If the thief, however, being particularly conscientious, both shuffles and heavily modifies the source code to disguise it, almost certainly the program either will no longer work at all, or it will be so unreliable that the thief will not find many buyers.

Assuming that, against all odds, the thief's source code shuffle has not induced massive internal problems into the rearranged software, the lack of knowledge of the software's inner workings will prove commercially fatal, either when the thief attempts to provide end-user support, or tries to amend the program to stay competitive. Experience shows that, even when changing one's own code, any modifications made to a modern program have a less than ten percent chance of working the first time and will usually cause some other part of the program to fail. The chances of successfully making modifications to code that was reverse engineered and lacks the high levels of abstraction explaining how the program works are probably on a par with those of winning a state lottery.

A software thief who lives by reverse engineering will die a death in the marketplace because of reverse engineering. The costs of reverse engineering, taken across the product's entire life, usually five to seven years, will invariably be higher than software written *de novo*.

As this paper has demonstrated, reverse engineering is the most expensive remedy of last resort. Software thieves have neither the time nor the funds to spend on massive reverse engineering. For them, the emphasis is on making a quick profit and moving on before the authorities track them down. The underlying reasons for reverse engineering are antipodean in nature to the motivation of a software thief who has no desire to take the time to understand how the software works when all that stands between him and illicit profits is the means to copy diskettes and photocopy manuals.

IV. Conclusion

Reverse engineering is a demanding and time-consuming process. Its results depend heavily upon a reverse engineer's skill and experience. The process has existed almost since the day that computers were created, being called into play whenever a programmer needed to understand how a program really works or what is causing an unexpected failure. Reverse engineering is not a technique used by software thieves because it takes too much time and costs too much money. Furthermore, even if thieves were to use reverse engineering, it would yield too much information of the wrong sort for a "fast buck" merchant.

The real issue is not whether or not reverse engineering should be a proscribed act because it might be used to misappropriate protectable

expression, but whether or not the large software companies can protect their ideas using copyright law. Opponents of reverse engineering are rarely programmers or small software development companies. Most programmers would vehemently oppose any restrictions on reverse engineering. They are united in their repugnance for those who plagiarize or steal the software of others. One must wonder why this is so.