# The Problem of Reverse Engineering

> *A common criticism of software publishers is that their EULAs prohibit reverse engineering, decompilation, and disassembly of their software. Software publishers typically restrict these activities because they risk exposing, and hence losing, to the public domain, the publisher's crown jewel-the secrets contained in the software's source code. Most purchasers of off-the-shelf software, however, care little, if at all, about the right to reverse engineer, and they certainly are not interested in paying more money to acquire this right. The entities that are most interested in acquiring this right are competitors of the software developer. A competitor should not be permitted to acquire the right to examine a company's trade secrets for the low price that the typical end user pays for the software.* (CK -- Footnotes omitted. Also, a EULA is an end user license agreement, typically typically shrink-wrapped and presented to the customer after the sale.)
>
> --- Robert W. Gomulkiewicz (a senior corporate attorney at Microsoft) and Mary L. Williamson (1996)

Many people misunderstand the nature of reverse engineering. Those misunderstandings shape corporate policies and legislation. As I've spoken to working software developers (especially and including testers) about Article 2B, I've been surprised to discover that we are as likely to misunderstand the breadth and importance of reverse engineering as the lawyers. (Article 2B is a 273-page proposed revision to the Uniform Commercial Code that will govern all software-related contracts. For more information, see my website, www.badsoftware.com, or my new book, *Bad Software*, which has a detailed appendix on 2B).

Originally, I presented material on reverse engineering at a conference on Article 2B for lawyers at UC Berkeley. That led to additional talks and to a paper for lawyers (Kaner, 1998). This paper is not about the legal issues of reverse engineering. And, though I mention Article 2B (as the key current effort to ban reverse engineering), this paper is not about Article 2B. The reverse engineering debate will go on even if we kill 2B. Rather, I have four objectives with this paper.

- First, to make you aware of a debate whose resolution will affect your work (you probably do reverse engineering quite often, and you might not like it if that gets banned).
- Second, to suggest some ways that you can articulate your concerns. (Your company might be one of the ones pushing for a ban on reverse engineering. Maybe you should explain what this would cost them if they succeed.)
- Third, to appeal to you for examples. I wrote this paper out of my own personal experiences. They're good examples, but there are better ones. A longer collection of good examples might carry a lot of influence.
- And finally, to solicit criticism from you. I'm going to make these arguments again and again and again and again. If there are holes or unfairnesses, I'd like to know about them. For that reason, even though I have adapted this paper from the one that I wrote for lawyers (cutting out legal arguments), I've left my descriptions of engineering issues largely intact.

## Background

Years ago, I worked in my parents' clothing business. We were mainly a retailer, but we also designed merchandise and had it custom manufactured. We used to buy samples from other vendors (and from our own suppliers) and take them apart to see how they were made. We did this to understand the quality of the products we were selling and the quality our competitors were selling. We did it to understand what types of problems our customers might have with a particular garment. We did it for training, to teach selected members of the staff how certain types of garments were made. And of course, we did it to understand how to knock off an improved version of a competitor's product. This was all legal. It's how many things are improved over time. But what we were doing was reverse engineering.

When a new machine comes to market, competing manufacturers will buy one and take it apart to see how it was built and what it does. This is reverse engineering. It is absolutely legal. It is a normal part of innovation, one of the foundations of continuous quality improvement within an industry.

There are inventions built into many types of products. When you take the product apart, you might come to understand the invention. But the invention can be patented. If it is original, it can be fully protected under the law, whether the competition understands it or not. And many aspects of the design of the product can be copyrighted or trademarked, so if they are original, they can be protected too.

Some software publishers have decided that their inventions are special (more special than everyone else's inventions, apparently) and that they should be able to prevent people from reverse engineering their products. As Gomulkiewicz and Williamson said, it is common to see clauses in software licenses that bar the customer from reverse engineering the software. Such clauses have been enforced in negotiated, non-mass-market licenses, but they have not been enforced in a software product that was sold (or "licensed") in the mass-market (see *Sega Enterprises Ltd. v. Accolade, Inc.*, 1992).

The result has been an ongoing debate over reverse engineering of software. One current forum for the debate is in the discussions of the proposed Article 2B of the Uniform Commercial Code, which will let publishers ban reverse engineering via "use restrictions" in the "licenses" that come with software products. Another forum involves proposed changes to the Copyright Act.

Even though some publishers and some manufacturers want to block reverse engineering (Apple and IBM have taken influential positions in this direction), much of the software community disagrees. For example, the American Committee for Interoperable Systems (which includes Sun Micro, Amdahl, AMD, 3Com and others) forcefully criticized Article 2B on the basis that it makes it much easier for software publishers to ban reverse engineering (Choy, 1998). So have associations representing software developers, such as the Institute for Electrical and Electronic Engineers (Reinert, 1998) and the Association for Computing Machinery (Gelman, 1998).

## What Is Reverse Engineering

As I understand the term, "reverse engineering" encompasses any activity that is done to determine how a product works, to learn the ideas and technology that were used in developing that product.

Reverse engineering can be done at many levels.

- At one extreme, you can study a product through strictly "black box" methods, feeding the program data (inputs) and monitoring its outputs. (“Black box analysis refers to reverse engineering techniques that do not involve copying or modifying the software.” Thomas Smedinghoff (1993, p. 85). A software license could specifically ban the use of a program in ways that are intended to reveal the underlying structure or technology of the program. Such a ban would work under Article 2B.
- At the other extreme, you can disassemble or decompile the program. In this case, you use a tool (such as a disassembler) to translate machine-readable 1's and 0's into Assembly Language, a low-level but human-readable programming language. Most programs are actually written in a high-level language (such as C or BASIC or COBOL). For example, you can issue a command,

  ```
  PRINT 5
  ```

  A translation of this simple command into Assembly Language might require several hundred (or more) lines of Assembler code.

  When you disassemble a program, you get thousands of lines of code, in a language that was not used by the original programmer, that lacks the comments, variable names, formatting and other signals used by the programmer to explain the meaning of the program. To translate this back into a high level language is painstaking work, and very time consuming. Andrew Johnson-Laird (1994) described this process in detail in his excellent paper, “Software Reverse Engineering in the Real World” 19 U. Dayton L.R. 843, 1994.

- Between the extremes, you can use various tools that reveal specific parts of the product without disassembling it (for example, it is easy to find the “strings”—the human-readable words—that are displayed by most programs) or that highlight behavior of the program that is not normally visible to the end user. This goes beyond black box analysis but doesn’t involve disassembly.

You can learn a lot by reverse engineering. But in terms of competitive use, there are limits on what you can do with what you learn. You can’t just copy your competitor’s code into your product—the copyright laws bar that. And many of the great ideas that you can find in code have been patented. You can’t use those either. As Johnson-Laird pointed out, reverse engineering is rarely a cost effective way of developing a competitive product.

## Why Do People Reverse Engineer?

I heard remarks on the floor at the NCCUSL annual meeting and in private discussions that programmers reverse engineer programs in order to discover trade secrets. That the programmer who learns these secrets will then be able to use the discovered technology to develop products that compete with the original program. The idea is that reverse engineering is a form of unfair competition, and therefore software developers should be allowed to protect their technology by banning the practice.

Andrew Johnson-Laird dealt with this misperception in his paper, "Software Reverse Engineering in the Real World”  (1994). He made the point that every programmer uses reverse engineering techniques for a variety of reasons. He provided hypothetical examples to illustrate his points.

My comments are intended to draw attention to, reinforce and extend Johnson-Laird's analysis. Unlike Johnson-Laird, I am not a specialist in reverse engineering. I don't offer reverse engineering services to clients. My examples are based on my personal experience as a software developer, software development manager, and software quality consultant. I chose examples out of my personal experience with the hope that by adding personal details, I could make them feel a bit more real to a reader who has probably never done this type of work. As you'll see, these reflect the normal course of normal practice, rather than the work of a reverse engineering hotshot (which I am not).

So why have my colleagues and I reverse engineered?

- Personal education.
- Understand and work around (or fix) limitations and defects in tools that I was using..
- Understand and work around (or fix) defects in third-party products.
- Make my product compatible with (able to work with) another product.
- Make my product compatible with (able to share data with) another product.
- To learn the principles that guided a competitor's design.
- Determine whether another company had stolen and reused some of my company's source code.
- Determine whether a product is capable of living up to its advertised claims.

Here are the examples:

### *Personal Education*

Professionals learn much of their craft by studying the work of other professionals. Lawyers read other lawyers’ briefs and depositions. Programmers read other programmers’ code.

I first studied how to write operating systems and interpreted programming languages by disassembling Applesoft, which was both a variant of the BASIC programming language and a disk operating system for the Apple II computer. As was to be expected from deep study of someone else's code, I also recognized (and took advantage of) opportunities to improve the product.

John Vokey (now the Chairman of the Psychology Department at the University of Lethbridge) and I used Apples to control experiments in our laboratories. We added various devices (clocks, analog-to-digital converters, tone generators, etc.) to these computers. To control them, we extended Apple's operating system. We also extended Apple's BASIC in order to analyze our data at a high level of precision. We published several papers detailing our analyses and extensions of Applesoft (For example, Kaner & Vokey, 1982a, 1982b, 1985).

We didn't do this work in order to compete with Apple. We extended the capabilities of this machine's software, making it much more useful to a research community.

The typical no-reverse-engineering clause doesn't contain exceptions for personal education or for development of extensions that will make the original product more salable. It just bans reverse engineering. Banning reverse engineering (via license restrictions approved by Article 2B) will mean a ban of these activities. Such a ban would have stunted my professional development and the development of many other of the best programmers that I know.

### *Limitations and Defects*

One of the oddities of the Apple operating system was that, at unpredictable times, it would insert noticeable delays between events. Our experiments presented stimuli for specific times and measured the times it took for people to respond. The Apple delays didn't matter to most users, but they created timing errors in our work. On reading the code, I discovered that these delays were caused by a clean-up operation--as needed, the operating system would reorganize its use of memory in order to make more memory available on demand for programs that needed it. The need to reorganize memory was an inherent limitation of the operating system. We were stuck with it. But we weren't stuck with the unpredictable timing. Once I knew the cause of the delays, it was easy to add a function to BASIC that let the programmer force a memory cleanup at a convenient time. Delays could now occur between trials (individual test cases within an experiment that might run a session of 100 trials) rather than, randomly, within trials.

One of the defects of this dialect of the BASIC language involved its generation of random numbers. A high quality random number generator (RNG) is essential for certain types of mathematical research. For example, simulations are sometimes the only way to compare the behavior (and thus the power) of different statistical functions under a wide range of alternative situations. (Kaner & Lyons, 1979; Kaner, Mohanty & Lyons, 1980). Errors in simulation could lead to misunderstanding of the characteristics of a function resulting in erroneous interpretation of experimental data. (For an example of this type of discussion, see Kaner, 1983).

John Vokey and I studied (sometimes by disassembly) the random number generating capabilities of several programming languages, including Applesoft BASIC. One common type of random number generator (the linear congruential generator) does a series of multiplications, additions and divisions to create each random number. The algorithm uses Integer arithmetic--fractions are discarded. Applesoft BASIC used what looked like the linear congruential algorithm but with floating point arithmetic (which kept the fractions) instead. The result was a generator that was less effective at producing a statistically random sequence of numbers. This is a subtle mistake. If you don't understand RNG's, your normal instinct as a programmer would be to use floating point arithmetic for these calculations. This error wasn't unique to Apple: Lyons, Vokey and I discovered it in two other programming languages as well. We developed a simulation-quality replacement generator for the Apple and eventually published it (Kaner & Vokey, 1984).

There's some sublety in determining that a random number generator (RNG) isn't working correctly, but simulations based on a defective RNG are usually invalid and usually have to be redone, often at substantial expense. Colleagues of ours, including some mathematicians, used Applesoft's RNG without realizing its defects. We initially recognized a problem in the Applesoft RNG by running some simple tests of it, but we didn't understand the extent of the problem until we read the (reverse engineered) code. A more subtle and more serious example of an RNG problem arose in early simulation work on 32-bit mainframes (machines like the IBM 360). Many mathematicians did serious work using a popular RNG called RANDU (Knuth, 1981). Simple tests did not and could not reveal RANDU's problems. The critical test that revealed problems in RANDU (the Spectral Test described by Knuth) depended on knowledge of RANDU's parameters--specific numbers used in calculations. RANDU's parameters were published, but to find the values of these parameters for other generators, we had to reverse engineer code.

So here we have a relatively common class of problems that aren't easily discovered except by reverse engineering. The consequences of incorrectly relying on the integrity of this function can be enormous--people model the behavior of many types of complex systems using random-number driven simulators. An enforceable ban on reverse engineering would prevent you (or your staff) from effectively checking the integrity of an RNG in a programming language you were using, killing your opportunity to prevent losses caused by the use of a defective RNG. Additionally, you might never discover that the RNG you are using is defective because an Article 2B-based ban on reverse engineering would prevent researchers who study such things from reverse engineering code, discovering a bad generator, and publishing a warning to the rest of the programming community.

Here’s another example of a defect. Year 2000-related defects are on the mind of many attorneys today. Article 2B would render enforceable a ban on reverse engineering or modification (repair) of the code by the customer. This is, after all, just a use and scope restriction. Result: a licensor who creates a defective program (such as Y2K-bugged) can prevent your company from determining the extent of defectiveness of the code and from fixing it. Instead, you have to pay the licensor top dollar for the privilege of patiently waiting until the licensor gets around to fixing your system. If you can find the licensor. And if the licensor is still willing and able to work with you. Hopefully the fix will be done, debugged, and working before January, 2000. It's bad enough to have an industry that routinely disclaims all warranties for its products. But to set up a legal structure that lets vendors routinely profit (and enforces their right to profit) from defects designed into their products? This goes a bit far, don't you think?

### *Defects in Third Party Products*

Over the years, I managed the development of several mass-market programs for contact management, desktop publishing, forms design, and project scheduling. To make these products profitable, I had to find ways to minimize the number of complaints we received from customers. At an industry average cost to the software publisher of $23 per call, complaint calls are expensive (for supporting data, see Kaner & Pels, 1998, Chapter 2).

- Our desktop publishing program had problems with a specific, popular brand of mouse. It worked with the other mice, but crashed and lost your work with this one. The vendor of the mouse didn't give us any help with this for a long time. Our customers bought their computers and their mice long before they bought our program. If their mouse caused our program to crash, our customers were more likely to demand a refund of our program than to trade in parts of their equipment. Our task was to figure out how our program and the mouse driver program failed to cooperate and redesign our program in a way that would work with the mouse. To do this without help from the mouse vendor, of course we had to reverse engineer the mouse driver. Article 2B would allow a ban on this. So under 2B, how *should* we have addressed this problem? By going out of business?

- Many printers advertised themselves as compatible with leading models such as (in those days) the HP LaserJet II, the IBM Proprinter or the Epson FX. Some of these claims were less than completely accurate. Result: our programs would work perfectly with the standard machine but fail with the "compatible." As with the mice, our customers bought their printers long before they bought our programs. We did a variety of tests (that could fairly be called "black box reverse engineering") in order to determine whether a popular printer lived up to the competitive claims made by its marketeers. And when we found incompatibilities, we had to find some way to make our product work with theirs. There are estimates that it takes 18 times as long for a customer to resolve a computer-related problem that involve two vendors' products as compared to a problem that involves just one product or just one vendor's pair of products. (Schreiber, 1997). Yet, for Company A to make sure that its product works properly with Company B's product, Company A is probably going to have to do some reverse engineering. A ban on reverse engineering, enforceable under Article 2B, would prevent vendors from doing the extensive compatibility testing and fixing that they do today. (*How extensive? When I worked at WordStar, we tested each significantly new version of our word processor with about 500 printers before releasing it.)* Such a ban would drive up your costs, as a user, enormously. And your system would be less stable.

Both of these are examples of reverse engineering done in the service of *interoperability*--making a product able to work with some other product, in this case by working around defects in that other product's software. Sometimes, when we recognized a defect in another company's product, we decided *not* to modify our code to be compatible with the other product. If the product operated strangely enough, if its market share was small enough, or if its technical support organization was arrogant and unhelpful enough, we would consider a decision to refuse to support the product. If we knew exactly how the product was malfunctioning, or how its performance deviated from a recognized standard, we could make an informed decision. After making that informed decision, we could explain the problem (the underlying defect) to technically sophisticated complaining customers. And we could privately explain it to magazine reviewers, who wouldn't then take us to task for choosing not to attempt to be compatible with the offending product.

### *Compatibility With (Ability To Work With) Another Product*

The previous examples involved reverse engineering to determine the cause and nature of a defect in a product. But sometimes we did reverse engineering simply to figure out how to work best with a product that worked properly. For example, when a printer or modem or other device would come out, some colleagues of mine would study the machine and its driver (a program that helps other programs control the device) to determine whether they should upgrade their programs to take advantage of new capabilities.

The documentation that comes with these devices is often quite modest. Many products come with documentation that was imperfectly translated into English from some other language. And the drivers often come with bans on reverse engineering. For example, I have a "Microsoft Software License" for the "Microsoft Windows 95 Driver Library." It says that "You may not reverse engineer, decompile or disassemble the Software." (According to the license, the Software is the Windows 95 Driver Library.)

So if the documentation is incomplete, inaccurate, and/or incomprehensible, how do you learn how to write code for a given device? The ban on reverse engineering might be intended to protect Microsoft (or the device makers who licensed Microsoft to publish their drivers in Win 95) from competition in the driver market or in the device market, but this ban would also block companies who merely want to make their product work with a device, companies that have absolutely no wish to compete in device/driver markets.

You can buy books on how to write device drivers. How do you think the authors of those books learned what to write?

### *Compatibility With (Ability To Share Data With) Another Product*

I assisted in the development of a desktop publishing (DTP) product and eventually became the project manager for several releases of it.

A DTP program takes text (from editors or word processing programs) as input and makes it easy for the user to design a page that will present that text effectively. The DTP program makes it easy to lay text down in columns, to wrap it around pictures, to apply various special effects to the text, and so on. Markets for desktop publishing and word processing programs have gradually converged. These days, I think the best word processors help the user with page layout about as well as a mediocre desktop publishing program. But back in 1990-1993, I would have defined the word "masochist" as someone who did fancy page layout with a word processor instead of a desktop publisher. Word processors and desktop publishers were complementary, not competitive, products.

Most word processors and editors save documents in proprietary file formats. To succeed in the DTP market, we had to figure out how to read, interpret, and preserve the formatting information in these files. For example, if you wrote a document with Word Perfect and applied boldface and italics to some text, you would expect that text to be italicized and boldfaced when you first viewed it with our desktop publisher.

It is possible to reverse engineer a file format. Imagine creating two copies of the same document, differing only in that some text is boldfaced in one file but not in the other. Look at the differences between the two files and you learn a lot about its coding of boldfacing. Repeat tests like this for every other imaginable treatment of text, for added pictures, etc., and eventually you can read the file. It is, of course, simpler and cheaper to just get a specification from the company that designed the file format.

I called many of these companies, asking them to send us their file format specs. Some agreed. Some didn't. For reasons that I'll never understand, one company not only refused; they said that I was trying to develop a competing product and then insisted to me that their license agreement barred us from reverse engineering anything of theirs, including their file format, and that we had better not try it.

I didn’t ask any of these companies for *permission* to reverse engineer their file formats. After all, I was going to be analyzing files that contained my documents, that I had created, that I held copyright to, that I could distribute freely to as many people as I wanted (none of whom had to sign any license agreements). But under Article 2B, this publisher *could* create a suitable restriction in a mass market product. All the license would have to say is that the user cannot use the product in a way that is calculated to reveal the file format. (Maybe the language needs some polishing, but you get the point.)

This might sound as though I’m stretching the point, but I’m not. I’ve met lawyers who think that such a restriction would be proper. One publisher's lawyer who attends 2B meetings privately explained to me that this type of restriction *should be* proper and that other companies could license the file format information from his company.

Of course, he admitted, this would leave him free to decide which companies to license this information to and which not.

Every company would like to be able to pick its competitors. But that doesn’t mean that the law should bless the practice.

We came into that market with no DTP or word processing history, facing established competition. A company much larger than us had about a 75% market share in our intended niche. We worked like mad to build a better product and eventually we overtook our largest competitor. This is the kind of success that Silicon Valley is made of. But if we hadn’t been able to read other companies’ data files, we’d have gotten nowhere. No one would buy a DTP program that can’t accurately import word processor files.

Silicon Valley became remarkably successful as an innovation-driven community because we all (engineers, not lawyers) were able to stand on each others’ shoulders. Changing those rules should be done with extreme caution.

## *Learn Principles That Guided a Competitor*

I worked as a human factors analyst / programmer for a company that made PBX’s (private telephone systems that you install in your offices). One of my tasks was to design an attendant’s console. This is the oversized telephone with way too many buttons that you can find on the desk of the receptionist in a typical 100-200 person company.

As part of my research, I spent a day or two at each of several companies, watching their receptionists work with other companies’ consoles (and sometimes playing with the consoles myself). My goals were to see the extent of variation of designs on the market, to figure out the principles that the other designers had considered important in designing their consoles, to see the ways that the consoles gave feedback (information) to the receptionists, and to see what made the receptionists angry or confused or impatient.

I learned far more from these observations that I could have ever learned from studying code.

Imagine yourself as corporate counsel for a PBX manufacturer who doesn’t want the likes of me studying its designs. Could you write a use restriction into the license for the PBX (or the software that drives the PBX) that bars a customer company from cooperating with a non-employee who wants to study the behavior of the PBX or any of its components? Of course. Should it be enforceable?

Next, consider doing competitive research on mass market software products. For example, imagine designing a desktop publishing program. Buy a dozen copies of each of your main competitors’ products. Sketch out some document designs (pen and ink drawings) and create some word processor files that can be used to provide text for the documents that people will create with the competitors’ products. Hire a few dozen temporary workers (specify a pattern of desired employment backgrounds to your local temporary labour agency). Assign a dozen people to each of your competitors’ programs and watch how they use those programs to create the documents that you have sketched out. For each design, measure how long it takes people to create the same document with the different products. And write down the errors that people make when trying to create these documents (patterns will show up and they'll be different for the different products).

Should a publisher of a product in the mass market be able to ban this kind of research? Surely, it could do so under Article 2B with an appropriately written use restriction.

### *Detect Theft*

At one company where I worked, a senior member of our staff joined a company that then published a product that competed with a key product of ours. Some people suspected that this new product had been built with some of our code, so our executives assigned some of our senior programmers to reverse engineering the competitor's product, looking for evidence that our most prized routines had been copied. We didn't find anything.

Other companies have found evidence of copying by doing reverse engineering. For example, in the case of *Apple Computer v. Franklin Computer* (1983), but a key piece of evidence was that James Huston's (one of Apple's programmers) name appeared in part of Franklin's code as did the word "Applesoft." Presumably, Apple found this by reverse engineering Franklin's code.

I don't see an exception built into Article 2B that allows a company to reverse engineer a competitor's code for evidence of theft. And if there is none in Article 2B, then *under what other law* do we find a rule that *in a licensing transaction*, one company has the right to reverse engineer a competitor's code?

### *Demonstrate Falseness of Claims*

Syncronys Softcorp published SoftRAM95. (I have no personal knowledge of the facts of this case.) SoftRAM95 was allegedly promoted as a software alternative to expanding your computer's memory. For example, according to the Federal Trade Commission's (1996) Complaint, Syncronys had claimed that its product would have the effect of doubling your computer's memory, from 4 MB to 8 MB.

According to the FTC, Syncronys sold approximately 600,000 copies of SoftRAM95. Andrew Schulman, a Senior Editor of O'Reilly & Associates, maintains a web page called SoftRAM95: "False and Misleading" at http://ftp.uni-mannheim.de/info/Oreilly/windows/win95.update/softram.html. This page is an archive of documents related to the SoftRAM95 case. One of the documents at that page is said to be a press release issued by Syncronys, that reported the results of a customer survey allegedly done by Dataquest, an internationally recognized technology market research and consulting firm. http://ftp.uni-mannheim.de/info/Oreilly/windows/win95.update/dataquest.txt. The results reported included these:

- "Eighty percent of customers surveyed said that SoftRAM 95 performed as expected.
- "Eighty-two percent described themselves as being either very satisfied or satisfied with the product.
- "Forty-eight percent described SoftRAM95 as being either their favorite or one of their favorite utility programs."

The only problem is that, according to the FTC's complaint, SoftRAM95 did little or nothing to increase the amount of available memory on a Windows 95-based computer. This case illustrates a big problem with high technology products: most people can't tell if they are being cheated.

Looking back to reverse engineering, one of the ways that the public learned that SoftRAM95 did (allegedly) almost nothing to increase available RAM was through reports of the results of reverse engineering. See, for example, Russinovitch's "Reverse-Engineered Disassembly of SoftRAM 95" and Schulman's "Analysis of SoftRAM 95." To the best of my knowledge, Russinovich and Schulman did this work as private individuals and not as part of any government investigation. Their work probably helped trigger a government investigation.

According to Charles Cooper, Dr. Dobbs Journal (which is a leading programmers' magazine) intended to publish a review of SoftRAM95 and was warned by Syncronys: "They [Syncronys] wanted to put us [Dr. Dobbs] on notice that they would protect their rights for defamatory or misleading statements as well as protect their copyrights and trade secrets." According to Cooper, Dr. Dobbs Journal chose to go ahead with the piece despite the warning.

How would this have played out if Article 2B was in force? Assume that the SoftRAM95 license would have contained a broad ban of all forms of reverse engineering. Could Russinovich and Schulman have done their reverse engineering research? Could a magazine like Dr. Dobbs Journal have published the results of reverse engineering studies without fear of liability? Would private citizens have been able to lawfully collect enough data on this product to interest the Federal Trade Commission in pursuing a case when most of the customers reported that they were satisfied with the product? Probably not.

## OK, Now What?

As you know, Software QA is changing to a new magazine. This column hasn't been carried forward, so here is a more general answer to the question of where we go from here.

***Regarding Article 2B:*** You might remember my initial column on Article 2B in 1996. In it, I laid out strengths and weaknesses of the proposed bill (which had been under development for about 9 years by then) and expressed some cautious optimism. Since then, progress has been difficult to achieve.

The two supervisory bodies of the Article 2B drafting committee both passed resolutions this year that urged a fairer balance of rights between sellers/licensors and customers/users. The American Law Institute specifically criticized the contracting approach (shrink-wrapped nonnegotiable contracts that you can't see until after the sale are a pretty unfair approach to doing business). The National Conference of Commissioners on Uniform State Laws passed a motion urging revisions to allow judges to invalidate contract clauses that interfere with competition, free speech, or fair use (such as reverse engineering). Changes were made to the latest draft of Article 2B to reflect these resolutions, but in my view, they were small-sized band-aids that did not begin to address the underlying issues.

Over the past month, the software development community has taken an oppositional stand to 2B. ACM and the Independent Computer Consultants Association have requested that 2B be tabled (which is politespeak for "cancelled"). The IEEE has recommended tabling unless certain issues (reverse engineering among them) are fixed (which probably won't be fixed). Several regional software quality groups are considering recommending cancellation of the 2B project. Additionally, the American Library Association, the Special Libraries Association, the American Association of Law Libraries, the Association of Research Libraries, the Magazine Publishers of America, the Motion Picture Association of America, the National Association of Broadcasters, the National Cable Television Association, the Recording Industry Association of America, the National Writers Union, the Newspaper Association of America, the Society for Information Management (which represents large customers), Consumers Union, and the Consumer Project on Technology (Ralph Nader) have all asked that the project be tabled. I have also submitted a detailed analysis, available from me by email (kaner@kaner.com), and requested that the bill be tabled.

It is possible that this collection of opposition will result in termination or reform of the project. For those of you interested in more detail or participation, the next meeting of the Drafting Committee is in Emeryville (near San Francisco) on November 13-15, 1998. I am teaching a seminar for software developers on what 2B is and how to participate in the process on November 12, at UC Berkeley's Boalt Hall (law school), room 105, from 6 to 9 p.m. This is not an official UC Berkeley function. There's no charge to attend. If you're going to come, please email me at kaner@kaner.com so I'll know to print a copy of the materials for you.

***Regarding Reverse Engineering:*** Whether Article 2B is fixed or not, the next five years will see ongoing discussion of reverse engineering. We really do need good examples and knowledgeable teachers, who can explain what reverse engineering is, why it's done, what it costs, and so on. There are many outlets for articles (written to educate lawyers) on the subject. If you can help, please contact me at kaner@kaner.com.

***Regarding Our Role in the Legislative Process:*** Several software quality advocates have been involved in Article 2B over the last three years. This includes James Bach, Doug Hoffman, Payson Hall, Watts Humphrey, Bob Johnson, Phil Koopman, Brian Lawrence, David Pels, Sharon Marsh Roberts, Johanna Rothman, Melora Svoboda, Clark S. Turner, Allan Wessels, and others (some people might not wish to have their names publicized). Our effect has been very significant. Even though the 2B Drafting Committee failed to adopt most of the proposals made by most of us, the cumulative educational effect of our active participation has been substantial.

When I first started attending these meetings, as the first small customer advocate and the first software quality advocate to start attending them, Article 2B was substantially worse than it is today and the Drafting Committee and the other participants were substantially more ignorant about software development issues. Those of us who have attended these meetings have changed that situation over the past three years. The worst parts remaining in 2B are there by design and will have their bad effects by design, not by accident because the drafters didn't understand the bugs they were creating. The debates are more technically knowledgeable and more technically influenced. And, partially because progress that we pushed for has not been made, this particular project is facing a serious prospect of cancellation despite the expenditure of years and (I believe) millions of dollars.

The legal system is just another system. It happens to govern what we do as professionals, so we have a professional interest in it. New laws are new products, to be integrated into an existing system. We have skills for analyzing new products in development, for finding areas for improvement, and for communicating the need for them. Those skills create an opportunity for us to play a valuable role in determining our industry's future.

As a technically knowledgeable person, you can get involved in this work at a technical level without ever playing games of Democrat vs. Republican. You can do it through the IEEE, ACM, or as part of an ad hoc coalition like the one we're still building on Article 2B. If you want to get involved in this type of work, please feel free to write me, Cem Kaner, kaner@kaner.com.

## References

*Apple Computer, Inc. v. Franklin Computer Corp.* (1983) *Federal Rporter,* 2nd Series, vol. P. 1240 (United States Court of Appeals for the 3rd Circuit)

Choy, P.M.C. (1998) Letter to the UCC Article 2B Drafting Committee, (American Committee for Interoperable Systems), October 7, 1998.

Cooper, C. (1996) "Syncronys warns Dr. Dobbs over upcoming review", *PC Week Online,* June 21, www.zdnet.com/pcweek/news/0617/21esync.html.

Federal Trade Commission (1996) *In the Matter of Syncronys Software*, Docket C-3688. www.ftc.gov/os/9610/c3688cmp.htm.

Gelman, L. (1998) Letter to the UCC Article 2B Drafting Committee, (Association for Computing Machinery), October 7, 1998.

Gomulkiewicz, R.W. & M. L. Williamson (1996) "A brief defense of mass market software license agreements" *Rutgers Computer & Technology Law Journal,* vol. 22, p. 335.

Johnson-Laird, A. (1994) “Software reverse engineering in the real world” *University of Dayton Law Review,* vol. 19, p. 843.

Kaner, C. (1983) “Results concerning Gatlin’s tests for bias in finite sequences” *Journal of the American Society for Psychical Research,* vol. 77, p. 31.

Kaner, C. (1998) "Article 2B and reverse engineering" *UCC Bulletin,* in press.

Kaner, C. & J. Lyons (1979) "Tables and power comparisons for different versions of the Kolmogorov-Smirnov and Schuster Statistics," *Technical Report No. 67,* Dept. of Psychology, McMaster University.

Kaner, C. S. Mohanty & J. Lyons (1980) “Critical values of the Kolmogorov-Smirnov one-sample tests” *Psychological Bulletin,* vol. 88, p. 498.

Kaner, C. & D.L. Pels (1998) *Bad Software: What To Do When Software Fails,* John Wiley & Sons.

Kaner, C. & J. Vokey (1982a) “Disassembling machine language programs without leaving BASIC”, *Compute!,* vol 4, issue #5 p. 146.

Kaner, C. & J. Vokey (1982b) “Modifying Apple’s Floating Point BASIC”, *Compute!*, vol 4, February issue, p. 68.

Kaner, C. & Vokey, J. (1984) "A better random number generator" *Micro*, vol. 72, p. 26.

Kaner, C. & J. Vokey (1985) “Subroutine Master”, *Nibble*, vol. 6, November issue, p. 46.

Knuth, D. (1981) *The Art of Computer Programming* 2nd Ed., volume 2.

Reinert, J.R. (1998) Letter to the UCC Article 2B Drafting Committee, (Institute for Electrical and Electronic Engineers), October, 1998.

Russinovitch, M. "Reverse-Engineered Disassembly of SoftRAM 95 - Windows 95 Version" http://ftp.uni-mannheim.de/info/Oreilly/windows/win95.update/softdiff.asm.

Schreiber, R. (1997) “How the Internet changes (almost) everything” *Internet Support Forum*, San Jose, CA.

Schulman, A. *SoftRAM95: False and Misleading*, http://ftp.uni-mannheim.de/info/Oreilly/windows/win95.update/softram.html.

Schulman, A. “Analysis of SoftRAM 95--SOFTRAM1.386” http://ftp.uni-mannheim.de/info/Oreilly/windows/win95.update/softram1.txt.

*Sega Enterprises Ltd. v. Accolade, Inc.*, (1992) *Federal Supplement*, 2nd series, vol. 977, p. 1510 (United States Court of Appeals for the 9th Circuit).

Smedinghoff, T. (1993), *The SPA Guide to Contracts and the Legal Protection of Software*, Software Publishers Association.