Export Controls, Encryption Software, and Speech

Statement for the RSA Conference, January 28, 1997

Dorothy E. Denning

Georgetown University

Three recent lawsuits -- filed on behalf of Philip Karn, Daniel Bernstein, and Peter Junger -- challenge the Constitutionality of U.S. export controls on encryption source-code software. A central claim in these suits is that the licensing requirement imposes an impermissible prior restraint on speech. As a programmer and computer science educator for more than 30 years, this makes no sense to me.

Software is, and always has been, the specification of a program that controls a computer; it makes the machine act with a given mathematical function. As such, software is an operational artifact. Export controls on encryption software are concerned with its operational behavior -- with the fact that encryption software loaded onto a computer is an encryption device. Export controls are not targeted at speech or ideas about the software. Given that the theory, formulas, and methods of any program can be expressed in a form that is not operational or easily made operational, I do not see how export controls on software restrict First Amendment speech.

A Short History of Hardware and Software

Early computing machines, such as Blaise Pascal's Pascaline in 1642 and Charles Babbage's Difference Engine in the 1820s, did not use software. The only functions they could perform were those built into their physical components. Although Babbage envisioned in the 1840s a general-purpose computer using punched cards to direct the operations performed, it was another century before the first programmable computers were actually built (Konrad Zuse's Z3 in 1941 and Howard Aiken's Mark I in 1944). Two of the earliest electronic computers, the Colossus, built in 1943 to break German codes, and the ENIAC, built in 1946 to calculate firing tables for the U.S. Army, were "programmable" with switches and pluggable cables. EDSAC, built in 1949 by Maurice Wilkes at Cambridge University, was a precursor to the stored-program electronic computers we use today. Programs were prepared on an external medium and then loaded into the machine's memory for execution.

The stored-program computer revolutionized computing by splitting the functions performed by a machine into two classes: *machine instructions*, which are embodied in the machine's architecture and provide the building blocks for all computation on the machine, and *software programs*, which are sequences of instructions. Software turned computers into universal machines -- the same machine could be used to compute an unlimited number of functions -- a concept envisioned by Babbage and later by Alan Turing. Software thus was invented for the purpose of extending the functionality of a machine so that it would be more useful. It was effectively a part of the computer that was easier to change than the machine's wiring, switches, or internal instruction set. Software and hardware are inseparable in any discussion of computers.

In the early days, programmers punched symbolic codes for instructions onto paper tape or cards. Special programs, called assemblers and compilers, were invented to convert these paper representations into machine instructions. Over time, compilers became more sophisticated, enabling higher-level languages for more compact descriptions of algorithms. A single statement in one of today's high-level languages is likely to compile into hundreds of individual machine instructions. Programs written in high-level or assembly languages came to be called "source code" and the compiled machine instructions as "object code." Functionally, there is no difference between source code and object code -- both specify the same function to be acted on by the machine. However, it is far easier to program in source code than in object code, and to let the computer do the conversion.

High-level programming languages allowed programmers to write software far more efficiently than would be possible if they had to program in machine code directly. Moreover, because the same source code could be compiled into object code for different or upgraded hardware, programs could be used on more than one machine, a feature called portability. These productivity-enhancing aspects of software eventually led to today's software applications, graphical user interfaces, the World Wide Web, and electronic commerce. None of this would have happened if programmers were still coding in machine language.

Today's programming languages have their roots in mathematics and symbolic logic. Even before the first electronic computer, the logician Alonzo Church devised a language for expressing mathematical functions and their computation. His Lambda Calculus influenced the design of early programming languages such as LISP and ALGOL, and later inspired the modern class of languages called functional languages. FORTRAN, which stands for FORmula TRANslator, was developed in the 1950s for computing mathematical formulae. It went on to become the dominant language for scientific computation. Another early language, COBOL, was designed to allow business programmers to express functional manipulations of business data in an easy notation.

I wrote my first program at the University of Michigan in 1966 using the Michigan Algorithm Decoder (MAD) language. The term "algorithm" in the language's name refers explicitly to the sequence of steps to be performed by a computer. For as long as I have been in the field, programming has been regarded as the process of writing algorithms in computer languages for the purpose of enabling a computer to carry out a certain function.

A discipline of programming, called software engineering, gradually emerged to cover program design, development, testing, and verification. The central concern of software engineering is that the designer can demonstrate conclusively that the software performs the desired function without error. The most powerful modern tools to assist the software engineer with this task -- such as theorem-provers, specification checkers, and predicate calculus transformers -- treat the software as undertaking mathematical functions on a computer. The argumentation and documentation required to convince other people that a software program works correctly is often much more extensive than the program itself -- few software engineers will accept a program by itself as a correctly functioning entity. The entire history of programming and software engineering has been pervaded with the notion that software directs the operation of a computer, and that it is essential to get that right. Computer science professors teach the standard practice of the field, which is that software is a precise functional specification that controls the operation of a machine.

Despite the shared use of the word "language," programming languages are quite different from natural languages such as English. Their purpose is not to communicate values, culture, emotions, feelings, political views, or arguments to a human being, or to coordinate action with another person. Rather, their central purpose is to encode the steps to be performed by a machine. Thus, whereas human languages are necessarily laden with ambiguity because words and phrases have real-world meanings which are subject to interpretation, computer languages are designed to be precise so that the functions will be performed correctly. Universities long ago recognized the fundamental difference between human languages and computer languages, disallowing use of the latter to satisfy language requirements.

Further, the process of compiling source code into object code is not the same as translating between natural languages such as English and French. One reason is that the result is intended for a machine, not a human being. Another is that the process is very different. The compiler assigns memory locations and registers in the central processing unit to instructions and data objects. It determines the precise order for directing computations in the CPU and the movement of data between the CPU and primary memory. It inserts instructions into the code to make use of routines already on the computer, for example, to compute a square root or display results on the screen. Most compilers also optimize the object code in order to speed up processing. Thus, any analogy between programming languages and human languages is extremely weak.

Mathematical Function is Infinitely Expressible

Software is not the same as speech *about* software. Programmers speak about their software all the time. They write descriptions and explanations of it, draw diagrams illustrating what it does, and prove properties about it. They express themselves in a natural language like English, mathematical notation, "pseudo-code," figures, diagrams, graphs, and many other forms of expression. They embed explanatory and descriptive comments in their source code -- asides that are ignored during compilation and execution.

The mathematics of programming teaches us that the functionality represented in any given program can be expressed in an infinite number of ways. The variety of expressions of the same function is so rich, that in practice teachers immediately suspect students who submit identical programs of cheating! But even more important to the discussion here: the functionality of a program can be expressed without a single line of executable code; it can be expressed as descriptions and equations using natural languages, mathematics, diagrams, and pseudo-code. This point is so crucial that it is worth emphasizing: *no program reflects an idea or function that cannot be expressed by other means*. The theory, formulas, and methods represented in software, whether source code or object code, can be expressed in a manner that is not easily made operational. Pseudo-code, which is cross between statements in a natural language and programming language, is particularly useful in this regard as it superficially resembles source code, but lacks the precision needed for conversion to object code. A skilled programmer can use this information to write a program that has the same effect, although reasonable effort may be required to do so. Thus, the notion that an executable source code program is necessary to convey an idea about a mathematical function to another person is not true.

Export Controls Do Not Violate the First Amendment

It is against this background that the contention that export controls on encryption software, particularly source code, are a prior restraint on protected speech, makes no sense to me. Export controls applying to particular functional artifacts hardly restrain a programmer's speech. All of the ideas and formulas represented in a program -- any program -- can be expressed in a non-functional manner. Export controls are not targeted at publicly available unclassified descriptive and explanatory information, mathematical formulas, or even pseudo-code despite the fact that this information can be used to produce a fully operational program that computes the exact same function as an export-controlled program. They also explicitly exclude educational information taught in college courses; fundamental research at universities; information concerning general scientific, mathematical or engineering principles; and information exchanged at symposia. Thus, export controls are not targeted at academic discussions about cryptography. The extensive number of academic conferences and courses on cryptography attest to this fact.

When I wrote my book *Cryptography and Data Security*, I included enough information about the Data Encryption Standard for a skilled programmer to implement the algorithm. I did this without providing any actual code and without violating any export control regulations. Indeed, I obtained my own information from a government publication, Federal Information Processing Standard (FIPS) 46. From this information, which included formulas, tables, and explanatory material, my students were able to produce software that correctly implemented the DES algorithm. (That they were able to do this does not mean that export controls are useless. Considerable effort is required to build complete and marketable products which meet user requirements when the software is not ready-at-hand.)

Thus, export controls on software are effectively controls on operational artifacts and not speech. Indeed, they are identical in purpose to those on encryption devices, which is logical given that encryption software loaded onto a machine *is* an encryption device.

The arguments above are valid regardless of the medium in which software is recorded. Nevertheless, the Administration has elected to require licenses only for programs in electronic form, even though printed code can be scanned into a computer. The rationale is that enough effort is required to convert source code in printed form to a functioning product that it is not necessary to license programs in print form, at least at this time. I can attest to this based on my personal experience a few years ago when a student tried, and failed, to produce a working version of the Data Encryption Standard from the source code in Bruce Schneier's book *Applied Cryptography*. Scanners and humans are error-prone, so unless one understands the program, it can be difficult to find and remove the errors. Perhaps this is why the students using my book succeeded, whereas this student failed -- he was never forced to fully comprehend the algorithm. Thus, in attempting to control only those artifacts that are easily used to encrypt, the government's distinction between electronic and printed materials is not irrational.

Larger Implications

I am concerned about the long-term implications of attempting to treat software generally as fully protected speech. Software has the potential of being highly destructive. Witness the Morris worm, computer viruses, or today's concerns about attacks on information infrastructure. Future viruses might someday bring down the power grid or direct the production of weapons of mass destruction. Do we really want to consider distribution of such software as free speech? Surely no one would say "logic bombs," or viruses should have the same protection as political or religious speech, even if an author claimed to be making a political statement. Yet treating software as fully protected speech could lead us down that path.

Export control regulations express judgments that exporting certain technological artifacts are harmful to the national well being and that the regulations make an important difference. It is reasonable and legitimate to question whether these regulations are serving the country. However, let us address that issue directly and squarely. Let us not muddle the issue by sweeping functional artifacts into the First Amendment. Free speech is one of our most fundamental and cherished rights. We should be cautious in applying it to the distribution of computer programs.