CONTROVERSY

Can UNIX survive secret source code?

Michael Lesk Bellcore

For secrets are edged tools
and must be kept from children and from fools.
– Dryden

Once upon a time there was a string processing language called COMIT. It competed with SNOBOL for certain kinds of text processing applications. Since even in the 1960s the proliferation of mutated versions of software was bringing chaos on users, the owners of COMIT decided there wouldn't be any modifications to it. Only binary was distributed; you haven't heard much of COMIT since. Ditto TRAC[1] (the currently popular language closest to TRAC is probably FORTH). Software is more attractive to the hackers if it can be changed. As more and more UNIX[2] suppliers restrict access to source code, fewer and fewer advanced research shops will be attracted to UNIX.

During the last ten years the fanatics of computer science departments have largely been in two groups: the UNIX camp, believing in modularity, and the Lisp camp, believing in integration. Why were these two systems so popular? Undoubtedly the hardware they used is part of the answer, as is their style; but surely the availability of source code was an important feature of their attractiveness to the universities. Unlike major manufacturer operating systems, the UNIX and Lisp machine environments had available source code without extravagant prices (for universities); they were small enough to be comprehended; and they ran on machines small enough that changes could be made without massive negotiations to get the approval of large

1. TRAC is a trademark of Rockford Research, if anyone still cares.
2. g/UNIX/s//the UNIX system/g

communities of affected users. This has resulted in both isolated improvements (a driver here and there) and massive rewrites (e.g. the Berkeley UNIX distributions).

I'm not saying that "information should be free" as some of the more extravagant university types do. People who write code, just like people who write books, are entitled to get paid for their work. But I think that the more restrictions that are placed on the use of code, and the greater the attempt to control exactly how systems are used, the less attractive a software product is, particularly in advanced computer research. And I expect, as a result, that there might well be more money to be made being slightly more forthcoming with source code.

Admittedly, circumstances alter cases. When railways were first developed, it was expected that they would operate the same way canals and turnpikes did: the railway company would provide only the right-of-way. Customers would show up with their own vehicles, and pay for the use of the tracks. It became clear quickly that this was intolerable and that a railway could not be operated safely without control over how many trains were operated, and in which direction on which track. Software does not carry analogous risks.

There are many reasons why suppliers do not wish to provide source code. For example, they may be concealing bugs in the hardware; it may be easier to program around certain conditions than fix them. This applies more often to turnkey products than operating systems, however, since when the customers write their own programs, unless the bugs are restricted to privileged instructions the users will find them. Another reason for not distributing source is to simplify maintenance. If the customers can't change things, the service staff knows what they will find. In the extreme I have seen a computer (intended for use in a factory) provided inside a heavy steel case with locks, and no keys given to the people who used it.

I suspect, however, that the primary reason for keeping source code secret is to preserve sales of both the hardware and the software. There are two arguments to be considered here. Source code might be kept secret to improve hardware sales (by tying customers who want a particular software feature to a particular piece of hardware), or it might be kept secret to improve software sales

(by preventing people from copying it or adapting it to make alternative versions).

First, if one cannot modify the source code to run on a different machine, anybody who wants a particular software feature must buy the hardware of the manufacturer who provides that feature. Thus, a customer trying to buy some software has to accept a particular model of hardware ("bundling"). Note that there is of course some risk in the strategy; the customer asked to buy something he doesn't want to get something he does want may decide to live without either of them, even though the software would have sold by itself. Usually the sale price of the software is so much less than the sale price of the hardware that the manufacturer is willing to run this risk. The conventional belief of hardware manufacturers that there is more profit in hardware sales than in software sales also pushes them to use software as a way to sell hardware.

This isn't, however, most of the answer. Until recently, only for operating systems written in source code could the customer take the software and run it on different hardware anyway, so that one would think that if the point of source secrecy was tie-in sales of hardware, those manufacturers with assembly language operating systems would have freely published their code. They didn't. Admittedly, this argument no longer applies, since clone hardware is becoming more common and no successful manufacturer can afford to ignore the possibility of imitations. However, even before there were clones, some manufacturers kept source code under wraps.

I even know one case where a particular UNIX utility was considered so valuable that it was going to be kept secret and provided only as binary for a specific machine. Over the next two years the plans to manufacture that machine fell apart and only a trivial effort was made to sell the hardware to the general market. When the time came to reconsider what to do with this particularly valuable program, all the managers involved had changed and the utility was now thought of such low value that the project was scrapped.

The major reason for keeping software secret, however, is just to preserve sales of the software. Software is easy to copy; the problems with computer games and low-level word processing

software are well known. Hence the desire of manufacturers to keep their product secret. But to me this is also not an overwhelming argument. In my experience, university departments generally do try to keep the contracts they sign; they do not steal software and pass it around. For example, academic researchers live by publication and dare not publish research done with stolen software.

Part of what is going on is simply fear of the unknown. There are a great many uncertainties about the future of the software industry. Understandably, producers wish to retain as much control as possible for fear that something bad might happen. The same phenomenon can be seen in other areas. For example, machine-readable texts are supplied, in general, only under extremely strict licenses. Where it is possible for a publisher to tie the text to a retrieval system and charge per access, they often try to do so. If traditional books were priced the same way you would have to pay a nickel every time you opened one. Part of the reason for the difficulty in getting typesetting tapes is that publishers are afraid of somehow losing all the revenue from their paper sales. We don't know what happens when machine-readable copies of conventional books are available nor what market and legal mechanisms are best for getting a fair share of the profits from new uses of material to the copyright holders. And, of course, while the uncertainty and the difficulty persist we do not explore the uses of new material as much as we should. I remember one spelling checking program which was switched from a dictionary-based word list to a newly made word list partly because of the difficulties of reaching any agreement with the dictionary supplier. Often a machine-readable file wanted is not available at all, even when a large sum is offered. I have known a publisher to turn down a larger sum for a machine-readable tape than the amount to be realized if everyone in the research lab's building had bought a copy of the book.

What is the history of UNIX in this regard? Bell Laboratories supplied it to universities for a nominal sum, with source code, and it became very popular as a computer science research tool. The availability of the source code was very important, as was the low cost and the practical advantages of UNIX over the other operating systems available for the PDP-11 series of machines.

The success of UNIX in the universities then caused it to become a valuable product elsewhere and to produce larger revenues for AT&T in the commercial market.

UNIX was then ported to some other kinds of hardware, starting with a Perkin-Elmer machine but most particularly including the Motorola 68000. This resulted in a number of manufacturers bringing out UNIX-based systems; some ported the code themselves, some hired firms to do it. AT&T arranged to make it possible for source code to be available for these other machines and also for the outside manufacturers to arrange binary licenses for those customers with no interest in the source code. As a result the UNIX market developed quickly and very effectively.

None of this would have happened if the source code had been kept secret, or if universities had been charged commercial licensing costs for experimental use. I believe as much in the technological imperative as anyone, but here accidental considerations of availability and ease of use were important. It is certainly possible to kill a market off by excessive regulation – how many nuclear power plants have been started lately, and how much have you heard recently about electronic funds transfer or videotext in the USA? One reason for the great success of the computer business was that people could write their own programs: it is a product for which the customer finds the use. The more the manufacturer constrains the uses, the fewer really advanced customers will be attracted.

Historically, secrecy hurts more people than just programmers. There is a story that during the Second World War, when millimeter-wave radar was developed, it was the first way of finding submarines running submerged with only a snorkel above the surface. The result was a sudden gain by the Allies in the naval war, and the Germans appeared to have no idea what had happened; submarines were captured fitted out with listening devices set up to try to discover how the Allies were finding them. When the RAF proposed to use the new radar in bombers, the Royal Navy objected: surely a bomber would be shot down, the Germans would find the equipment in the crashed plane, and then the gains at sea would be reversed. The argument went up to Churchill, who ruled in favor of the RAF; a month or so after outfitting the aircraft a bomber was shot down and the Navy

prepared for the end of the good hunting. To their amazement nothing happened at sea; the German submarines appeared to continue in ignorance. After the war it turned out that the German Army had studied the bomber that was shot down and they didn't see why they should tell the Navy about the unusual radar set they had found on the plane.

I would like to compare the source-code versus binary-code choices with the drama and the cinema. In the traditional dramatic theatre, playwrights write plays, which are then produced and directed by a variety of companies, perhaps centuries later. As a result we get to see many interpretations of the same text, and people today discuss the actors they have seen in particular roles. Opera is similarly enhanced by new interpretations and new singers. By contrast, for movies the screenplay belongs to the producers of the movie; the movie can be remade only with their permission. Now there is no shortage these days (or in earlier times) of remakes of movies. But are any of them any good? In the few cases where the second version of a movie is better than the first, it is usually a movie derived from a literary work anyway, so that the second movie is really taken from the original book or play, rather than the first movie (e.g. "Oliver Twist"). If you try to maintain too much control of your work, you stifle further improvements. As an extreme, in Shakespeare's day dramatic companies did not want texts of plays published, hoping that nobody else would be able to put them on. We must all give thanks that this particular form of secrecy was not sufficiently thorough or effective to keep us from having copies of Shakespeare's plays.

You may object that the drama may permit more creativity in performance, but after all the movies make more money. So let me talk about some more commercial enterprises. When foreign equipment rules in the telephone business changed, there was an explosion of answering machines and similar devices. This has resulted in an increase in the number of calls completed. The service may be worse (many customers feel annoyed at paying for a call when they only get to talk to a tape recorder) but the revenue from phone service is up. By contrast, in Europe, the national PTT organizations combine dominance of service with the power of legal regulation and limit entry into the communication

business. As a result, for example, cable television is less common and the total choice of television channels is much fewer than in the USA.

Again, much of the problem is uncertainty. Music publishers are willing to sell sheet music for popular songs, rather than sell only recordings, since the copyright laws restrict other performances without permission and thus the sale of sheet music does not threaten a flood of unwanted imitation recordings. No such confidence is available today in the software business. Thus the owner of a successful operating system must worry that other people will copy and imitate his software and then argue with a judge that it was new work.

The recent development of "look and feel" suits is adding to the problem of uncertainty. Nobody knows what is protectable, legally: is it what is patentable (specific new ideas), what is copyrightable (the lines of code), or the "look and feel" of something? The more doubt about what can be protected, the more tendency to keep things secret in the hope of compensating for the inadequacies of the legal system. This does not mean that we should make everything protectable no matter how trivial; extremes of protection, as in some of the "look and feel" claims, inhibit further technical progress and the development of standards.

Providing source code to universities is thus, in my mind, an effective way of advertising one's product, getting other people to develop applications, and producing a trained set of users who will try to buy the same system when in new jobs. It would be even better if the legal problems could be eliminated: that is, if public copyright law, rather than trade secret license agreements, could provide protection. There are many in universities (and elsewhere) who find the negotiations involved in obtaining a trade secret license very frustrating, with the lawyers on both sides insisting on variations in wording that technical people can not understand. Even better than copyright might be a simple technical fix that reassured the sellers without inconveniencing the users; I don't know what that might be. As an example, if CD-ROMs became a standard way of distributing software, they are hard to make locally and cheap to mass produce. Thus it is likely that people would be encouraged to buy the information rather than steal it. Unfortunately, since a CD-ROM is no larger than a

modern fixed-head disk, it is not difficult to transfer the information. It is also important that the price of software be reasonable; and as the number of computers in the world increases, I gain hope that more and more suppliers will aim for many sales at low prices rather than a few sales at high prices (regretfully, this doesn't seem to be happening in the CD-ROM market).

Now, my optimism that cheap distribution of source code to universities will be repaid with more sales at higher prices to commercial customers, as happened with UNIX originally, may be irrelevant. The commercial world clearly is moving towards more secrecy. So what will happen next? University customers and research groups, in particular, will ask for access to source code. And as the standard UNIX suppliers now include more and more companies whose source is not supplied, I predict more and more interest in universities for finding alternative operating systems.

Another change, at the same time, is the rise of multiprocessor computers. The tendency over the last ten years for CPU chips to get cheaper more rapidly than they get faster has made it advantageous to build multiple processor machines, and at present we do not really know how to take advantage of them. Advanced software for these systems is still being built. The opportunity for a new operating system is there.

It is important that at the same time the AI community, which has long relied on Lisp code from MIT that was also available in source form, now finds itself faced with various Lisp machine manufacturers who have made significant extensions and changes they are keeping confidential. As a result AI researchers too are perhaps a fertile ground for a new environment.

Finally, UNIX itself is getting old now. The conventional date for the start of UNIX is 1969, so pretty soon it will be twenty. There are UNIX programmers younger than the code they are using. By comparison, can anyone imagine, in 1970, using a 1950 operating system? Maybe computer science as a whole is slowing down, but I certainly hope the field has some new operating systems still to come. UNIX is also now so big that it is hard to change and as frustrating in many ways as any conventional operating system; to quote Dick Haight, once we had manual pages, now we give master's degrees in *stty*. Presumably *ioctl* is worth a Ph.D.

For all these reasons, I think the universities are ripe to start with a new operating system. I don't know what it will be, but we need good support for multiple processors, networks of workstations, and graphics. A good system should also be small enough that it can be comprehended and used for teaching. John Lions wrote of sixth edition UNIX that it was the only powerful operating system whose documentation fit in a student's briefcase; in later editions this has been fixed. And, of course, the system must be efficient enough for real jobs to be run on it.

Consider, for example, the problems of partitioning a UNIX program to run in a multiprocessor environment. Right now there are two obvious breakpoints: a subroutine call or a `fork()` call (i.e. a shell command). Compared to IPC, subroutine calls are very fast, and shell commands are very slow. So subroutine calls are used very freely in programs, whereas shell pipelines can not be (if each one costs a second, decent response time requires that only a few processes be started). Thus, although breaking programs apart at commands is easy to do, since the processes on either side are already well separated and there are very few of them, the scarcity of processes makes it impossible to get a big speedup by dividing a program in this way. What about cutting at subroutine calls? Today these are usually 100 times faster than interprocess communication, and thus it is not practicable to separate processes at this point. Furthermore the shared memory discipline makes it hard to divide programs at calls.

One possible answer is "lightweight processes" that will not require such high overhead. An alternative would be a level of "task," with limits on shared memory and rapid startup. But it is necessary to find some coupling point whose cost is comparable to that of the IPC invoked when the processes communicate across the coupling. Consider a railway train. The capacity of a freight car (goods wagon for you UK types) is measured in tons, 50 to 100 tons being typical in North America. It would not make sense to try to put together a train of cars each designed to hold a hundredweight; too much of the weight would be in couplings and wheels. But articulated unit trains carrying thousands of tons in long strings of cars designed never to be separated aren't catching on either. The capacity of the cars must be substantially larger than the weight of the couplings, and the complexity of the

couplings must be minimized (there is still no electrical cable that runs end to end on a freight train, although there is an air pipe). But the units should still be dividable to permit adjustment of train capacity to offered load and rearrangement of cars for routing.

In computer terms, this suggests that if too large a fraction of the time of a process is spent communicating it will not be efficient. There is an alternate view that process overhead doesn't matter because extra CPU boards can be devoted to handling communications. Hence, even if 2/3 of the processor were just I/O boards, what difference would it make? Although the logic is reasonable, my worry is that effectively the cost of such CPUs will be high enough to make intelligent process separation worth more effort.

Of course, this raises the issue of what IPC mechanisms should be provided. Realistically, one would like them simpler. Comparing UNIX files with the IBM file systems available at the time, and looking at the old *mpx* or Berkeley sockets, one dreams of something concise and neat.

Can we bridge the gap between "integrated environments" and "modular tools," so that AI and other CS groups may share software? That, in itself, would be a major step forward in US computer science. I have some hopes because the rise of neural-net research raises the CPU demands of AI groups, increasing the chance that they will wish to make common cause with other computer scientists.

So, I am looking for some computing system that runs on multiple small machines, supports graphics and AI work, and remains comprehensible. Are there any systems around today that meet all these requirements? I don't know one. But I suspect that somewhere, in an attic, somebody is building such a system. Maybe it will look like UNIX, and maybe it won't. But if it is to be successful, it should have readily available source code. And in current trends that means it won't be a variant of UNIX.

What might the feel of the new system be? It would, above all, have a sense of efficiency and compactness. We have a large supply of elephantine operating systems, whose documentation

overflows not just a student's briefcase but his backpack.[3] We need another system people can understand. Remembering how well UNIX got along without indexed sequential files and PL/I compilers, we must aim for "that supreme gift of the artist, the knowledge of when to stop" (Sherlock Holmes). Thus the system we want will not come from any large team carefully funded by a corporation or big government to produce "the next UNIX." Any such team will have too many people, be answerable to too many funders, and wind up putting too much in the system. The goal is not to anticipate and provide for all needs; the goal is to permit the users to do this for themselves. Tom Sawyer did not plan how he would paint the whole fence; he got others to do it for him. In addition, a small system can be built by a small team; if it is done in a "skunk works" it is more likely to appear so trivial to the owners of the code that they will let it out cheaply, and then complain about it later when they realize what they have done, as have some AT&T lawyers. The more important that the project is as it is planned, the more likely it is to be kept secret. If it is a success, it will be kept secret because it is so valuable; and if it is a failure, it will be kept secret to conceal the bad news. *Vide* Star Wars.

So we can not look for the next operating system. We can only expect it. Somewhere, perhaps in a physics or chemistry lab (or an English or music department), it will come. If we don't recognize it when it does arrive, we will ignore it and it will die; but if we do recognize it, we'll wrap it up in confidentiality and it will die anyway. Is there any hope? Carlyle wrote that if Christ returned today people would invite him to dinner and make fun of what he had to say. Our best hope is that the next operating system arrives as a joke.

[*submitted Sept. 16, 1987; revised June 3, 1988; accepted June 6, 1988*]

3. The truly technology-driven will presumably suggest that the answer is higher-reduction microfiche.