

John M. Cox

Mr. John M. Cox

English 125.068

10 October 2003

Introduction to a Critique of Proprietary Software

Computing is at a point of crisis. It has, for most of its history, been reclusive – originally because of cost, and more recently because of complexity – but this period of limited exposure waned in the late 1990s. Today it is all but over, at least in the industrial world. Originally created during the Second World War as a tool for decrypting Axis communiqués, the first computers were house-sized monstrosities attended by armies of technicians, and, for all their size, orders of magnitude less powerful than anything available today. They were unimportant beyond their military applications. Today a computer goes to school with most inbound college freshmen, and though it may literally have less volume than a breadbox, it can encrypt email with a cipher more powerful than the most complex of German military codes. Computers are fast becoming a standard home appliance, and although the most recent US census shows them appearing in only 51% of American households (Households), their penetration pattern is similar to that of VCRs and color televisions, which are almost universally present (Myth). Indeed, the fact that this article makes use of online sources presupposes technical availability among scholars, and speaks to their social influence and ubiquity.

For technologists, at least, it sounds as if the future of computing is anything but imperiled. If computers are becoming more of a social force, whence the crisis? Briefly, the problem is one of commercialization, which is fundamentally at odds with the continued improvement of computing. To explain this, some terms must first be defined.

Computers have two basic components: *hardware* and *software*. *Hardware* refers to the physical parts of the computer – the central processing unit which is analogous to the computer’s brain, the random access memory that is analogous to its short term memory, the hard disks that are analogous to its long term memory, and the numerous other components that have less obvious analogues to the human body. *Software* refers to actual programs that the computer runs. Microsoft Windows is a piece of software, as is the money manager Quicken. The problem with these definitions is that, in practice, hardware and software are not nearly so clearly distinct: any piece of hardware can also be made as a piece of software. This, in fact, was how the earlier computers operated: a program would be made as a hardware module, which, when plugged in to a computer, could then be used. This practice is no longer common in consumer computers, but it is still standard in other industries. The computers in cars, for instance, are all hardware. Programs found in hardware are now called *embedded systems*. When I say *software* in the remainder of this article, I am using it in the more conventional way – as a shorthand meaning *computer program*. The software I am discussing could, however, easily be found in an embedded system.

The crisis brought on by commercialization is one of software, not hardware. It is an indisputable fact that, as far as hardware is concerned, industry has been almost entirely a boon. The rapid pace at which the speed of computer processors increases – roughly doubling every two years, an observation which technologists call Moore’s Law (Moore’s) – is due to the enormous industrial progress that commercialism and capitalism make possible. However, these social and economic systems have pernicious consequences for software. Specifically, in a system which treats everything as a product, and in which all products are the property of some individual or institution, intellectual freedom and the freedom of information are at a

disadvantage. In order to understand why this is true for computer programs, which, on their face, look exactly like any other project, another explanation of technological terminology is in order.

When computer programs are written, they are first in the form of *source code*. Source code contains the instructions that the programmer gives to the computer. For example,

```
for ( $i = 1; $i <= 5; $i ++ ) {
    print $i . "\n";
}
```

is a snippet from the Perl programming language¹ which instructs the computer to print

```
1
2
3
4
5
```

Real programs are made up of many lines of source code. However, when an end user runs a program, that end user is usually not interacting with the source code at all. Instead, they are interacting with a *binary file* (also simply called a *binary*), which is the form in which most programs are distributed. Computers do not speak human language. They do not even speak the language found in source code. It is commonly said that computers speak in ones and zeroes, which is to say that they speak binary code. This is not strictly true; computers really speak in voltages. They are, after all, electrical machines. The numeral *1* in binary code means high voltage, and the numeral *0* means low voltage. Binary code is a representation of this, and binary files contain the language the computer uses to decide when to generate these voltages². The

¹ Strictly speaking, Perl is an interpreted rather than compiled language. That distinction is beyond the scope of this discussion; suffice it to say that interpreted and compiled languages are significantly alike for the purposes of this discussion that the difference between them can be ignored.

² Technically, binary files contain *machine language*, which is binary code that varies depending on the type of computer on which the software is run. The binary file for Quicken, for example, is different on a PC than it is on an Apple computer. This is because the central processors for those two types of computers use different instruction sets. A more specific discussion of instruction sets and machine language is beyond the scope of this essay. What is important is that computers actually execute binary files rather than source code.

important distinction here is that computers execute the binary file and not the source code itself.

All that a user needs to run a program, then, is the binary file. For consumers, this is all that matters. For technologists, however, and especially for computer programmers, the source code is far more important than the binary. For one thing, binaries are produced from source code by a process called *compiling*. Most technologists have no problem compiling any piece of source code to produce binaries. Second, and more importantly, the binary shows what a program can do, while the source code shows how a program does it. This is vitally important to technologists and programmers, who are often more concerned with questions of *how* than questions of *why*. If the source code is available, questions of security, efficiency, and operation can all be answered. If, however, only a binary is available, these questions cannot be easily addressed.

This brings us to our point of crisis. Software companies are, before all else, companies, and must be primarily concerned with financial matters. They consequently have a pronounced economic interest in restricting access to source code. After all, if that code is available, anyone with expertise in computing can create a binary file without paying for it. For this reason, most commercial code is *proprietary*, or *closed source*. This means that product is the binary, and the source code is usually not available. In the rare cases where it can be viewed, it is legally encumbered in ways that prevent users from creating binaries or writing derivative works. By contrast, many technologists and programmers prefer a system of *open source*, where the product in question is the source code itself and not the binary. In the open source paradigm, it is still possible to charge for software, but the source must be as available as the binary. Many of the most important innovations in computing, including most of what makes the Internet itself run, were created under the open source framework.

Our crisis, then, is that as computing moves more and more into our public sphere – a place which is dominated by commercial and capitalistic interests – the open source structure is threatened³. Though there are many companies that have embraced open source practices, including IBM and Apple, there is significant economic interest (and, consequently, legal pressure) working against the open source model. Open source maintains intellectual freedom and the freedom of information, but at the cost of a decreased ability in our current economic and legal frameworks. Closed source provides the opposite. The question, then, is whether we, as consumers of and, increasingly, participants in technology, prefer economic or intellectual gains.⁴

³ This threat is not merely theoretical. Numerous companies have used the courts in an attempt to stifle open source development. The most stunning example came in 2003 when the SCO Group (<http://sco.com>) filed suit against computer hardware and software giant IBM over the latter's support of particular open source projects. The fight spills over into the public sphere as well. Industry publications such as Forbes magazine (<http://forbes.com>) regularly ridicule open source proponents as Communists, thieves, or somehow un-American. See, for example, Daniel Lyons's article of 14 October 2003, in which he likens the image open source proponents to that of "proles linking arms and singing the 'Internationale'" ([Lyons](#)).

⁴ This essay is indebted to the materials made available by the Open Source Initiative (<http://opensource.org>) and the Free Software Foundation (<http://gnu.org>). In the interest of full disclosure, the author is an open source developer.

Works Cited

Households with Computers, 2000. The Public Policy Institute of New York State. 2001

<<http://www.ppiny.org/Reports/jtf/Table%2013.htm>>

Lyons, Daniel. "Linux's Hit Men." Forbes. 14 October 2003

<http://www.forbes.com/2003/10/14/cz_dl_1014linksys.html>

Moore's Law. Intel Research. 17 June 2003

<<http://www.intel.com/research/silicon/mooreslaw.htm>>

The Myth of an Emerging Information Underclass. The Cato Institute. 20 November 1997

<<http://www.cato.org/dailys/11-20-97.html>>