

Opening the sources of accountability

by Shay David

Abstract

This paper scrutinizes the concept of accountability in light of free and open source software. On the view that increasing accountability grants value to society by motivating those most likely and able to prevent risk and harm to do so, I argue that while developing software collaboratively, licensing it openly, and distributing its source code freely are promising first steps in the long journey to rehabilitate accountability in our highly computerized society, our very understanding of what accountability is changes too. This paper analyzes the concept of accountability in an open environment and explores the implications in two mission-critical application fields in which software plays a significant role — electronic voting, and electronic medical records. It further considers the potential remedies to accountability's erosion that free and open source software offer, and the ways in which accountability can be generalized to collective action if we understand it less as punishability and more as a culture that encourages the prevention of risk and harm. With such reconceptualized accountability in mind, I find that code visibility, a self-imposed standard of care and sensible licensing arrangements, are a potent, practical, and effective alternative to the strict liability standards offered as a solution to the accountability problem by earlier scholars.

Contents

[Introduction — The co-production of accountability and software](#)
[The sources of accountability in computing](#)
[The effect of open source on four barriers to accountability](#)
[Open source and accountability in practice](#)
[Further considerations to opening sources of accountability](#)
[Conclusion](#)

Introduction — The co-production of accountability and software

Accountability, like many other virtues, is an elusive and hard-to-define concept when considered in the context of ever-changing information technology. Like other esteemed

values, accountability's definition is inseparable from the moral groundings we choose to espouse — the meaning we assign to accountability depends directly on our understanding of concepts like responsibility, fault, and guilt which further depend on our conception of individuality and our theory of causation. Accordingly, the ranking of accountability in one's scale of revered values depends on the valence and efficaciousness that she attributes to these concepts. In the literature we can identify three main lines of moral and practical reasoning that render accountability as an imperative: (1) accountability as a virtue that is desirable in its own right; (2) accountability as a guideline for answerability which motivates precautionary behavior that, in turn, caters social welfare; and, (3) accountability as a tracing tool that allows us, *a posteriori*, to identify the people involved in accidents and damage-inducing errors, punish the responsible if necessary and compensate the victims, if possible. We should note that within all three lines of reasoning accountability is context-specific — any call for accountability is a call for a specific arrangement of social relationships which, as we know, are increasingly mediated by technology. Moreover, in recent years information technologies in general and computer and software technologies particularly, have played a paramount role in defining our sense of individuality, privacy, security, and power and are, therefore, of great consequence for our understanding of virtues like accountability. Accordingly, the key line of investigation in this essay seeks to explicate the relation between accountability and software technology in the context of a paradigm shift from traditional software development to free and open source software systems.

As we know, technology does not work in a social void — it is shaped by the society within which it operates, and it shapes this society in return. In fact, we can think of technology and society as co-producing each other in a process that shapes our social values as much as it molds our technological artifacts [1]. This is not to say that such co-production is straightforwardly explainable or that the ethical implications of this process are easy to digest or even clearly understood. When thinking about ethics in general and moral virtues particularly in the context of information technology, as much as it is sometimes hard to admit evolving technology often implies that we need to rethink some basic concepts. Deborah Johnson and James Moore discuss how when considering ethics in the context of computers and software we easily find ourselves in a vacuum of rules, and we soon need to apply our norms in uncharted terrain (Johnson, 2000; Moore, 1985). But while such extension of rules is often useful, if we perceive our task as one that involves fitting new technological pigeons into known pigeon-holes we run the risk of overlooking important moral possibilities that are offered by novel technologies.

With this in mind, in what follows I argue that in the case of accountability we should not simply stretch our application of known moral dilemmas to the new situations that arise as the degree of computing increases. Instead, departing from earlier accounts that simply warn against appalling accountability loss, what I suggest is that our conception of accountability in matters technological should change where a new set of development methodologies — that I will denote here with the catch-phrase FLOSS (free/libre open source software systems) — play a part. We should note that as a culture we hold accountability in high regard and work towards a fair and effective principle of it, although our motivation is driven by multiple and sometimes contradictory impulses; specifically, we are struggling with how to hold designers accountable for what happens with their devices, software notwithstanding, and this struggle imposes several theoretical and practical difficulties. However, with changes in the manner and principles of software design, particularly with the rise of FLOSS, we have an opportunity to rethink some of those difficulties, to see if the tenets of accountability are better suited by the new approach and to

rethink the very concept of accountability and how it plays out in socio–technical practice.

These new development methodologies under which software is developed collaboratively depart from traditional, romantic, notions of authorship and personhood and from economic models of firms and markets and to this extent change the foundation upon which our conception of accountability rests. In the discussion below I will briefly survey the relevant aspects of FLOSS history [2], examine in detail several examples that show how openness becomes a key feature for overcoming barriers to accountability that are otherwise hardly surmountable, and argue that they sidestep the problem of generalizing accountability to collective action as it is present in traditional development environments. Based on a detailed analysis of two types of software errors, and on observations of two environments in which software plays a key role and in which FLOSS recently made its marks (electronic voting and electronic medical records), I conclude that accountability can be upheld in a computerized society if we forfeit our understanding of it as punishability and understand it as a culture that motivates collective action towards the prevention of risk and harm and accepts self–endorsed standard of care.

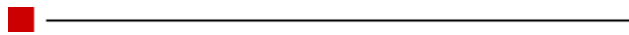


The sources of accountability in computing

Several years ago in her thoughtful paper "Accountability in a computerized society," Helen Nissenbaum argued that while effective attention was given to computer systems' reliability and safe design the "conditions under which computer systems are commonly developed and deployed, coupled with popular conceptions about the nature, capacities, and limitations of computing, contribute in significant measure to an obscuring of lines of accountability" (Nissenbaum, 1996). She further warned that "Unless we address these conditions and conceptions, we will see a disturbing correlation — increased computerization, on the one hand, with a decline in accountability, on the other" (Nissenbaum, 1996). Other authors share the same view [3]. Her argument goes on to investigate the moral and legal aspects of the term, and identifies four barriers to accountability: (a) The problem of many hands–responsibility, which is usually analyzed in terms of a single individual, does not generalize to collective action and, therefore, in a computerized society responsibility is thinned because in software it is customary for many people in many organizations to work collectively towards the end product; (b) The problem of inevitable bugs–glitches in the performance of software are viewed as inescapable and, consequently, a dangerous approach has developed that frees developers of their accountability; (c) Views of the computer as scapegoat — the increased agency we tend to assign to non–humans absorbs the human accountability when the computer is blamed for faults and the investigation stops at that; and, (d) Ownership without liability — the vocal debates on intellectual property rights (IPR) obscure the equally important discussion on the responsibility and liability that should be associated with those rights. As a conclusion Nissenbaum calls for two measures: an industry–wide espousal and implementation of an explicit standard of care for software development and deployment, and the adoption of strict liability as a main legal standard in the litigation that follows software–related accidents.

Nissenbaum's account is profound and it gives a good first approximation of what accountability in a computerized environment is and useful insights into how the software

world had traditionally handled the issue. But while I agree with her first call-for-action, I reject the second, based on reasoning that finds strict liability unjust in the specific case of software, as I specify below. What is more, I find that while her arguments on the four points that erode accountability are quite correct for software in general, the rise of FLOSS methodologies, which have recently attained maturity that renders them qualified for mission-critical deployments [4], offers us an opportunity to revisit the ‘four problems’ and reconsider them anew. FLOSS turn the ‘many hands,’ ‘inevitable bugs,’ and ‘ownership without liability’ arguments on their head, and offer an insight into the ‘computer as scapegoat’ argument and its implications for human agency, and hence accountability, too.



The effect of open source on four barriers to accountability

Inevitable bugs, exactitude in software and accountability

On the face of it, in our society that highly esteems rationality and that increasingly resorts to technology to solve problems of ‘inefficiency,’ one would expect that the failures that software systems sometimes exhibit would be intolerable. Surprisingly, however, software is judged by another standard: after all, would we accept it if our cars would sometimes turn themselves off in the middle of the highway or if our refrigerators would change the selected cooling temperature on their own spoiling our food, or if electrical power would cease often and unexpectedly? Probably not, and yet when computers exhibit the parallels of those failures (*e.g.* crashing in the middle of a customer presentation, erasing important documents, or sending off malicious software to our e-mail contacts without our knowledge) we are not as alarmed — software seems to be perceived as error-prone and in that respect conceptually different from other types of technology. Generally speaking, in the software world people have come to the conclusion that bugs are inevitable [5]. In part in order to reconcile this double standard, and in part because technological systems are generally perceived as deterministic, predictable, efficient, and fault-proof, we resort to biological metaphors — like bugs and viruses — to describe software glitches. However, the use of the ‘bug’ metaphor to cover the entirety of modeling, design, and coding errors is misleading and needs to be unpacked.



**Bugs are hard to detect.
It is often very hard to
even know when errors
and failures occur, let
alone to analyze why
they occur or to actually
fix them, but should this
deter us from assigning
accountability?**



What exactly is a 'bug'? The word choice evokes the image of little living entities that are present where they shouldn't be and are hard to get rid of. It also implies that 'bugs' are something that we have to learn to live with as they are simply part of the world. Moreover it obscures the fact that software bugs are produced by people (designers). And indeed in actuality, as we well know, bugs are hard to detect. It is often very hard to even know when errors and failures occur, let alone to analyze why they occur or to actually fix them; but, should this deter us from assigning accountability? The definition of 'bugs' depends highly on the definition of human expectation from a computerized system to begin with. A recent survey of directions of software quality explains:

"A software failure occurs when a piece of software does not perform as required and expected. A software fault is a malformation whose execution causes a failure, and an error is a flaw in human reasoning and performance that leads to the creation of the fault. When examined closely, we see that a software failure is a deviation of the execution of a program from its intended behavior. Thus, the notion of a failure (and by implication, the notion of an error) is a relative one. Failures and errors are deviations from intent. Without a clear notion of what was intended for a program's behavior, it is not possible to be sure that it has strayed from that intent, and therefore that a failure and an error have occurred." (Osterweil, 1996)

In other words, defining bugs is inseparable from defining human expectations regarding the correct operation of the software. But fully defining intent and correctness turn out to be very knotty undertakings. Brian Smith (1985) argues against the very possibility of proving software's correctness, a practice to which a whole engineering discipline is devoted. One of the first problems that arise, Smith notes, is that correctness and intent are abstract, subjective concepts that can be interpreted in many levels, and that therefore have no objective meaning. It is not at all clear whose intent counts in the analysis of a failed operation. Smith explains:

"Suppose the people want peace, and the President thinks that means having a strong defense, and the Defense department thinks that means having nuclear weapons systems, and the weapons designers request control systems to monitor radar signals, and the computer companies are asked to respond to six particular kinds of radar pattern, and the engineers are told to build signal amplifiers with certain circuit characteristics, and the technician is told to write a program to respond to the difference between a two-volt and a four-volt signal on a particular incoming wire. If being correct means doing what was intended, whose intent matters? The technician's? Or what, with twenty years of historical detachment, we would say should have been intended?" (Smith, 1985)

Smith goes on to make the point that any attempt to define the correct operation of software will be stifled by an even more significant theoretical barrier: correctness is conceptualized as a correlation between the specification and the program. But the specification is a formal representation of intended functionality that, just like the program, takes into account a model of the world, and not the world itself. It is exactly the model of the world, which is arguably an abstraction that is necessary in order to write the specification and the program, which in fact constitutes a barrier between the world and the software. Any test or simulation that contrasts the software and its specification will only compare two formal systems with each other *vis-à-vis* the model, but will hardly guarantee the 'correctness' of

the software operation in the real world.

Consequently, we should distinguish between at least two classes of bugs. The first class covers software failures in situations where the expected execution is clear, at least in terms of the model. Such bugs are usually caused by bad programming or faulty design. For example, consider bad coding of the software's variable memory management. A program runs in the computer's memory space and allocates memory as needed. When memory is no longer needed it is discarded, or 'freed,' only to be used later. An error in the memory allocation mechanism might have far reaching consequences because the program fills the memory with 'junk' data, and bad memory assignments can cause the software to use 'crazy' values in its variables and produce unpredictable results or even 'crash' [6]. A parallel of this class of bugs in, say, automobiles, would be a design or a manufacturing process that results in a car for which under certain conditions the water system can jumble with the electrical system. In both the case of software or of such a badly designed car the expected performance is clear, but an internal error prevents the proper execution. Other bugs in this class include infinite loops (the program halts forever) and memory leaks (the program's performance degrades and slows with time until it crashes) among others.

The second class of bugs involves problems in design and malfunctions in performance in face of unpredictable or changing environmental conditions in which the very essence of what the intended functionality should be is unclear (and if we accept Smith, we realize that these situations might be more common than we thought). To be sure, such software malfunctions may actually arise in exactly the situations in which the designers' model of the world falls short in some significant way of the world itself. Smith tells the story of the American Ballistic Missile Early Warning System that mistakenly interpreted radar signals reflected by the rising moon to be a Soviet attack. Such lunar reflections were simply beyond the scope of the model that was used by the system's designers. Another classic example of a bug of this class is the infamous Y2K bug from which many earlier computer systems presumably suffered. These so-called vulnerable software systems might not have had any bugs of the first class, but because their designers did not foresee the problems that would arise from resetting the year digits of date-related fields to '00,' their performance at the turn of the millennium was unpredictable. The parallels of this class of bugs in the automotive industry are found in designs of earlier cars that were built with simple airbags, and that when faced with today's environmental conditions (like traffic density) and modern safety standards, are considered unsafe and can exhibit unpredictable accidents. In sum, bugs in this category include situations in which real-world data affects the performance of the system in ever-changing ways which are not accounted for in the design. In Nissenbaum's terms, these are the bugs that should be classified as 'natural hazards.'

I argue that these 'natural hazards' are at the base of the erroneous conception of bugs as inevitable, and that the distinction between the two classes is important because the mechanisms in which such bugs should be exterminated are different and accordingly the accountability we should attribute to their producers is different. While bugs of the first kind can usually be spotted and corrected by a meticulous code review without having to run the program, it is the bugs of the second kind which are inevitable in traditional models of software development, and which extermination needs a different approach. Nancy Leveson, who describes system safety in terms of emergent properties that arise only when certain systematic conditions are met, suggests an integrated approach to safety that integrates system engineering, software engineering, cognitive psychology, and organizational sociology (Leveson 1995a, 1995b). While Leveson's approach can systematically reduce hazards and is surely a step in the right direction, extensive simulation

and testing hopelessly remains a staple of most attempts to clean bugs of the second type in the traditional software paradigm. Hopeless, that is, because, as this analysis clearly shows, under no circumstances could complex software be fully tested in lab conditions. The only way to fully test composite software is to run it in the real world, or as Sancho Panza, Don Quixote's devoted squire, once insisted, we can inquire the frying pan and the eggs all we want, but at the end of the day "the proof of the pudding is in the eating" (Cervantes, 2001).

People that fail to acknowledge this truism but realize that the key to success is to test the software in 'real world' scenarios keep coming up with ever more extensive frameworks for testing and simulation, which, of course, never guarantee results but are likely to bloat the budget of any project [7]. Such attitudes bring to mind another fable, one of Borges's extraordinary tales, which seems to well describe the futile hopes of fault-proof software testing. Borges tells the story of an empire that once had a penchant for uncompromised exactitude:

"[I]n that empire, the art of cartography attained such perfection that the map of a single province occupied the entirety of a city, and the map of the empire, the entirety of a province. In time, those unconscionable maps no longer satisfied, and the cartographers' guild struck a map of the empire whose size was that of the empire, and which coincided point for point with it. The following generations, who were not so fond of the study of cartography as their forebears had been, saw that that vast map was useless, and not without some pitilessness was it, that they delivered it up to the inclemencies of sun and winters. In the deserts of the west, still today, there are tattered ruins of that map, inhabited by animals and beggars; in all the land there is no other relic of the disciplines of geography." (Borges, 1998)

And just like in Borges's outlandish empire, testing and quality assurance methods only lead to more testing techniques and quality assurance frameworks that are later abandoned because they fall out of favor [8]. The problem with all these testing paradigms, of course, is that they only test the model, but don't test the world.

When thinking of accountability in this context we can draw the following conclusions. First, we cannot hope for the same type of accountability for both types of bugs. Bugs of the first type are more preventable, and therefore we can expect a higher level of accountability from the people responsible for them. Bugs of the second kind, however, are in principle harder to pin down and we should not expect that the people responsible for them will be accountable in the same manner. Coming back to our earlier example, do we really want to hold the inventors of early computer systems accountable for the Y2K bug that occurred some three decades after their systems were introduced? Should we rebuke the missile warning system's engineers for failing to account for lunar reflections? I argue that we don't, at least not in the common sense of accountability. Although these pioneers might be deemed blameworthy when using the yardsticks of fault and causation [9], I argue that we should not hold them accountable in the same way as those producing bugs of the first type. The justification for this is simple: competing with accountability is the value of technological innovation and progress. This paper is too short to encompass the debates over innovation itself; suffice it to note that technological innovation as an explaining force for techno-social interactions is a problematic concept for those, like this author, who reject technological determinism, but at the same time we cannot ignore the general sense in which our culture, since the Enlightenment and the scientific revolution, has come to valorize a state of constant technological flux. On this view, had the inventors of early

software systems concerned themselves with solving the Y2K bug or designing for every possible effect of every celestial constellation, they would have probably not been able to get any working system to begin with; instead they would have been busy to this day in sketching the software parallel of Borges's map.

For related considerations I reject the call for strict liability as a standard for litigation in software-related accidents [10]. While in some cases liability surely should be assigned, it should be assigned *a posteriori* and on a case by case basis. Establishing strict liability as a standard in order to balance the potential harm and encourage safe design would actually throw the baby away with the bath water. It would in fact deter individuals, firms, and development communities from pursuing mission-critical software development for which, by definition, the critical nature of the mission implies a high "cost" when things go awry. We will never know how many people have been saved by software-controlled radiation treatment machines during the years that their operation was still "unsafe." Likewise, we can never know what the world would have been like if the department of defense had delayed the deployments of its strategic software systems until they were bug-free. We can be sure, however, that if the designers of these software systems and other "high-risk," mission-critical software had needed to focus on strict liability consequences instead of focusing on developing safe software, innovation would be stifled to a large degree. There needs to be a balance between technological innovation and safety, and this balance will be best served by a balanced principle for assigning liability. Strict liability is by no means balanced. It would never favor innovation in this equation and we should, therefore, need to part with the idea of using strict liability as a standard for such situations. This leaves us with Nissenbaum's other suggestion: to establish a high standard of care, an idea which I strongly support. This is where we can learn our first lesson from FLOSS.

Many hands and many eyes

Published in this very journal several years ago, Eric Raymond's influential manifesto [The Cathedral and the Bazaar](#) calls for a radical change in the way people design, build, and use software. Raymond anatomizes a project called "fetchmail," that he ran as an experiment for the new theories concerning software engineering suggested by the development model of Linux (Raymond, 1999). He looks at the ways software is written, debugged and used while contrasting two development styles: (1) the traditional "cathedral" model, exercised by most of the commercial world, under which software is written by firms in a hierarchical manner; and, (2) the "bazaar" model of the Linux world whereby software is built collaboratively, by groups of self-appointed volunteers. Raymond argues that one key difference between the models is the relation to the debugging task:


" ... every problem "will be transparent to somebody" ... the person who understands and fixes the problem is not necessarily or even usually the person who first characterizes it ... But the point is that both things tend to happen quickly. Here, I think, is the core difference underlying the cathedral-builder and bazaar styles. In the cathedral-builder view of programming, bugs and development problems are tricky, insidious, deep phenomena. It takes months of scrutiny by a dedicated few to develop confidence that you've winkled them all out. Thus the long release intervals, and the inevitable disappointment when long-awaited releases are not perfect. In the bazaar view, on the other hand, you assume that bugs are generally shallow phenomena — or, at least, that they turn shallow pretty quick when exposed to a thousand eager co-developers pounding on every single new release. Accordingly you release often in order

to get more corrections, and as a beneficial side effect you have less to lose if an occasional botch gets out the door. And that's it. That's enough." (Raymond, 1999)

Following the insights of Linus Torvalds, the originator of the open source operating system Linux, Raymond formulates a law that states: "given enough eyes every bug is shallow" (Raymond, 1999). Or, alternatively, "debugging is parallelizable." Software development is a process that involves many hands, but open source makes this putative disadvantage into its greatest advantage. If we are bound to live in a world where the complexity of systems calls for a large-scale collaboration — in fact it dictates that any reasonably complex task be done by more than a few people — then open source makes this disadvantage into an advantage by hedging the risks through collaborative effort. Unlike in traditional software models where the users only get an opaque block of bits in executable form, in FLOSS models the programs are distributed to the users with their source code and the users can use this source code in order to improve the software or find bugs of the two types we discussed. By reviewing the code they can find bugs of the first kind, and by testing the software in their own particular environment, they can find bugs of the second kind. The Open Source Initiative Web site explains:

"The basic idea behind open source is very simple: when programmers can read, redistribute, and modify the source code for a piece of software, the software evolves. People improve it, people adapt it, and people fix bugs. And this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing." (Open Source Initiative, 2004)

In regards to accountability, collective action is the key concept here, where the joint goal is to prevent risk and harm, and the public good is safe and reliable software. Such action works at two levels: what is escapable from the eyes of one programmer or software designer is inescapable from the eyes of another; people can find bugs of the first kind just by looking at the source code that other people wrote. Second, by releasing many versions of the software as *early* as possible, testing is done in the real world and not in simulated environments. This guarantees that the maximum amount of possible code paths be executed and, at the same time, it guarantees that as many external (environmental) variables are internalized, since what constitutes a far-fetched operation scenario for one person is the everyday state of affairs for another. In Brian Smith's terms, we could view this software evolution to be in effect a perfection of the model that mediates between the software and the world. As more people test the software in real-world environments, more discrepancies between the abstract model and the world are found (regardless of whether the source of the discrepancy is a lesser initial model or a changing environment). The model can be improved, and the software can be continually adapted to changing scenarios [11].



**If we are bound to live in
a world where the
complexity of systems
calls for a large-scale**

**collaboration ... then
open source makes this
disadvantage into an
advantage by hedging
the risks through
collaborative effort.**



In sum, in open source projects one cannot escape accountability while resorting to explanations of inevitable bugs. Surely, some bugs would remain (mostly of the "natural hazard" kind), but generally bugs are treated as "shallow" phenomena that can be done away with, rather than acceptable defenses or escape-hatches from answerability for the consequences of one's actions. What is more, instead of having "many hands" erode accountability like in traditional software development, in the open source model many hands can participate in coding and debugging, while many eyes ensure that accountability is maintained or even expanded to include "collective accountability." While a self-selected development community is not a well defined legal entity that could assume responsibility or compensate victims, I argue that it still facilitates communal accountability which is larger than the sum of its would-be personal-accountability parts. Beyond the accountability of each member, the developers' community as a whole can be thought of as accountable for the software it produces. The common use of documentation and tracking technologies such as Concurrent Versions Systems (CVS), that document changes and contributions to all open source projects, perfectly serve the need of answerability and allow for the generalization of accountability from the individual to the group level. The reputation of individuals within a group and that of groups within a larger community is what drives open source development to pursue excellence in performance. The high level of scrutiny that such openness facilitates ensures that for groups of developers that, to a high degree, are motivated by personal reputation, (a) a culture of responsibility is developed, if only because one can fix the errors of another; and, (b) the motivation (and ability) to prevent risk and harm is increased, not by the fear of punishment, but rather by the desire to maintain one's standing within their social group (by writing good code). To be sure, both of these effects on accountability are triggered not by legal regimes but by social structure [12]. Can such measures, tough, ensure performance adequate for mission critical applications? As the case studies below clearly show — they do, but first let us explore the remaining barriers to accountability.

Ownership without liability

Existing software producers want to eat their software cake and leave it whole. That is, developers want to enjoy the rights of ownership without having to pay the associated costs, and — surprisingly — they often succeed. The discourse of rights versus duties traces back to the philosophies of Thomas Hobbes (Hobbes, 1651) and John Locke (Locke, 1821) who were both concerned with "natural" rights and duties and their implications for social interaction and the power of government [13]. In recent years, however, and in large part due to the unprecedented expansion of capitalism (often at the expense of state power), these debates have shifted away from a discussion of the "natural" rights of property and

dangers posed by the sovereign and morphed into a legal and ethical debate about the conflicting interests of consumers and producers. As part of his account on professional ethics in information systems, Davison notes:

"Software producers claim that they have the right to protect the fruit of their endeavors — the software programs. Furthermore, they have the right to be compensated for the resources that they have expended in the development of that product. ... To protect their right to the fruit of their endeavors, they claim that consumers have a duty both to pay the price and to respect the intellectual property contained within the product — by not stealing it. Notwithstanding these rights and duties, consumers may claim that they have the right to use a product for which they have paid, and indeed have the right to expect that that product will be free of defects (bugs). This imposes a duty of quality (and professionalism) on the developers of the software to ensure that the software is indeed bug-free. Consumers expect that a product be competitively priced, providing value for money." (Davison, 2000)

As Nissenbaum noted, these conflicting rights and duties are not well-balanced. A trend has developed according to which software producers use their excessive power to

"demand maximal property protection while denying, to the extent possible, accountability ... This denial of accountability can be seen, for example, in the written license agreements that accompany almost all mass-produced consumer software which usually includes one section detailing the producers' rights, and another negating accountability." (Nissenbaum, 1996)

Most IPR advocates take for granted two premises: (1) that consumers and producers are two distinct groups with conflicting interests; and, (2) that intellectual property, like other types of property, requires some level of protection from appropriation. On this view, it is not surprising that the software producers will try to erode accountability. If consumers are a resource to be exploited and software is property that needs to be protected, the logic of capitalism implies that increased accountability can only mean reduced profits. To change this imbalance, Nissenbaum calls for strict liability, hoping that the producers will change their ways and produce better software when they realize that their profitability might be at risk if their "property" malfunctions and they have to pay damages to consumers. As we have seen above, this is not a tenable solution. Again we can look to FLOSS for a more potent alternative. FLOSS methodologies challenge both premises of IPR debates: at least to some degree they collapse the dichotomy between producers and consumers, and they refuse to accept the idea that software should be protected in the same ways as physical property (Nissenbaum, 1995; Stallman, 2002; 1987).

Richard Stallman, founder of the Free Software Foundation, explains *why software should not have owners* in this way:

"Digital information technology contributes to the world by making it easier to copy and modify information. Computers promise to make this easier for all of us. Not everyone wants it to be easier. The system of copyright gives software programs 'owners,' most of whom aim to withhold software's potential benefit from the rest of the public. They would like to be the only ones who can copy and modify the software that we use ... But if a program has an owner, this very much affects what it is, and what you can do with a copy if you buy one. The

difference is not just a matter of money. The system of owners of software encourages software owners to produce something — but not what society really needs. And it causes intangible ethical pollution that affects us all. What does society need? It needs information that is truly available to its citizens — for example, programs that people can read, fix, adapt, and improve, not just operate. But what software owners typically deliver is a black box that we can't study or change. Society also needs freedom. When a program has an owner, the users lose freedom to control part of their own lives." (Stallman, 2002)

Noticing that much of the existing relationships between consumers and producers are negotiated through software licensing agreements, the Free Software Foundation turned the copyright system against itself and created GNU [14] with the General Public License (GPL) as an alternative to traditional copyright. The process of publishing software under the GPL, which became known as "copylefting," ensures that the software will always remain free, and avoids potential threats of delayed copyrighting by interested parties that might have occurred had the software been simply released in the public domain [15]. According to its architects' logic, naming this legal hack "copyleft" is very simple: while proprietary software developers use copyright to take away the users' freedom, free software uses copyright law to guarantee it. Hence, this reversal justifies the reversal of the name from "copyright" to "copyleft." Copylefting, according to this same logic, is a vital first step in the withering away of the intellectual property system as a whole, as Moglen argued in the pages of this [journal](#) (Moglen, 1999).

Using licensing regimes such as the GPL, FLOSS frameworks circumnavigate the barrier of "ownership without liability" not by accepting the burden of liability but, in contrast, by forfeiting ownership rights. The licensing agreements they advocate empower consumers, instead of limiting their rights. In fact, by allowing the consumers (that is, users) free access to source code, FLOSS blurs the line between consumers and producers. As we have seen earlier, in such an environment accountability transpires from collective action. In sum, with FLOSS we now have a new situation: accountability without ownership.

The computer as a hybrid scapegoat

A forth barrier to accountability arises in situations in which computers, as non-human agents, mask human responsibility for errors. As "smart phones" become smarter, "expert systems" gain expertise, and computerized "decision support systems" actually make decisions for us, the problem of accountability in light of failure dramatically worsens. Unfortunately, we are all very familiar with those situations in which finger pointing is directed to a screen of one machine or another, not to a human [16].

Open source does not offer a clear solution to this problem, but recent analyses of the ways in which open source developer communities work might give us hints to an interesting ontological possibility. Nicolas Ducheneaut (2003) tackles the problem of human versus computer agency with analytical tools originally developed by Bruno Latour's (1987) and Michel Callon's (1998) actor-network theory (ANT). In line with ANT's treatment of material and non-material actors symmetrically, Ducheneaut's investigation into the development process of open source demonstrates the blurring of boundaries between the activities of human and material actors. Consequently, he suggests that we should view an open source project as "essentially a hybrid network — a heteroclite assemblage of human and non-human actors, entangled in specific configurations that may vary over time" (Ducheneaut, 2003). In open source, "design" and "use" are principally indistinct

from one another. By extension, we can reason that the hybrid–network approach is an adequate description of the relationship between humans and computers, not only throughout the development phases, but during "use" as well.

Open source advocates seem to be very self–aware of this intertwining of human and material agency. In the latest release of the Apple Computer's Macintosh Operating System X (an open source system) when a particularly severe bug occurs (*e.g.* when an application crashes) the computer "reacts" to anticipated allegations of fault by automatically uttering in a metallic voice: "It's not my fault." This tongue–in–cheek approach does little to ameliorate the errors themselves, but it does raise users' awareness to issues of fault and causation. In his recent series of digital prints entitled *OK/Cancel*, the artist Perry Hoberman captures the irony present in Apple's self–mockery. Hoberman presents imaginary screen shots of computerized error messages that could be real. One especially astounding image is shown below (Hoberman, 2003).

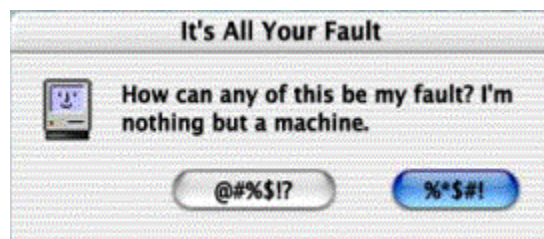


Figure 1: *It's All Your Fault*, digital print from the series *OK/Cancel* (Hoberman, 2003).

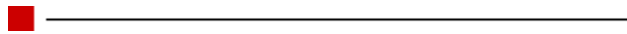
Works like Hoberman's speak to the frustration users feel when they encounter dead–end situations, which evolved as we gave software the agency to make decisions for us. Hoberman ridicules the situation by assigning the computer even more agency — not only the ability to perform actions, but also to feel angst. The computer, in anticipation of its owner's scorn, asks sadly — "how can this be my fault? I'm nothing but a machine." Irony aside, however, the "computer as scapegoat" mindset remains a severe problem for accountability.

The task of exploring the full moral and ethical implications of actor–network theories (ANT) deserves a separate paper and would be too lengthy a detour for this article. Suffice it to note here that the consequences for accountability might be remarkable. If we accept such actor–network views, then "computer as scapegoat" arguments should not be perceived as problematic. Arguably, if we accept that material objects have equal agency as nodes of the network that underwrites the software world, we have to accept cases in which computers (or any other material actors for that matter) simply are responsible for error, and not a stopgap that obscures some sought–after human actor. Accepting actor–network theory, however, has significant implications to our understanding not only of machines but of humans as well. Equating material and non–material actors implies depleted agency for humans, a loss which might be wrongly used to justify accountability's erosion. We are all familiar with visions of horrific brave new worlds and matrices in which humans that have lost their agency eventually lose their ability to control the machines. Any ANT that assumes material agency has a heavy burden to lift in order to show that it does not end in dystopia.

One way of avoiding the slippery slope of full–blown ANT is to take from it the common idea that material and non–material actors work as hybrid networks, and yet resist the

corollary that implies that all actors are equal in every respect. As it happens, the weakest edge (or node) in a hybrid network might well be a non-human actor, but this does not imply that humans lose control or, in turn, lose accountability. With this, the yardstick of causation becomes unnecessary when assigning accountability. In certain cases, a computer might truly be the cause for error, and yet accountability can be maintained.

"Accountability" in this case means a software design that allows investigations into the causes of errors and an examination of the relationships between actors. The question becomes not "who/what caused the error?" but rather "what did the human actors do in an error-prone environment to prevent risk and harm?" and "how can this error be used to improve similar systems in order to improve the welfare of society?" When asking these questions, and by framing the meaning of accountability in this way, it is evident that open systems have a higher degree of accountability than closed systems. In open software systems, it should be easier to assess the exact events that lead to an error, to evaluate the performance of those individuals involved, and to learn from their mistakes in the hope of preventing similar harm in the future.



Open source and accountability in practice

After looking at the theoretical aspects of four barriers to accountability and understanding the potential remedies offered by FLOSS and FLOSS's implications for the reframing of the term, we can now turn to look at some practical examples. We will see how in electronic voting systems openness resuscitates a sense of answerability that in closed systems can be lost due to ownership-without-liability and computer-as-scapegoat barriers. We will also examine how in the health care sector, unlike the appalling existing situation of closed source software licenses that conflict with federal regulations, open systems can easily satisfy contradictory needs, while, at the same time, render bugs as far from inevitable.

Case I: Open source voting

As the 2000 U.S. presidential elections clearly showed, the existing punch-card technology used in many ballots around the U.S. is highly inefficient, not trustworthy, and in urgent need of improvement. The call for action advocated by industry evangelists and politicians alike suggests abandoning the error-prone [17], paper-based mechanisms, and transitioning to digital voting systems. The new systems, however, introduce a set of new problems that mainly revolve around issues of fault tolerance, trust, and the risk of deliberate manipulation of election results (vote-rigging) by insiders and outsiders alike. While all three problem categories raise concerns of accountability, it is the third type that deserves special attention since it makes a mission-critical electronic voting system even more critical. Unlike in the case of other software systems that are normally evaluated on their ability to minimize errors, here the system should not only be fail-proof — it should be manipulation-proof as well. Peter Neumann, who moderates the Association for Computing Machinery's (ACM) RISKS forum, analyzes the risks involved and proposes solutions:

"The risks for vote-rigging on COTS [commercial off-the-shelf] systems include these: (a) someone tweaks the BIOS [built-in operating systems] of the voting machines; (b) someone tweaks the OS [operating system] of the voting machines; (c) someone tweaks the applications code; (d) someone tweaks the

compiler.

(a) Can best be dealt with via physical security only — have non-flashable BIOSes, and disallow unauthorized access. The rest require both a publicly available Open Source codebase, and physical security to make sure that what you think is on the machine, actually is. And that the right OS has been installed and the right compiler used." (Neumann, 2002)

Neumann's call for publicly available open source code can be explicated in terms of accountability. It is an example of a situation in which accountability should be understood primarily as a system of accounting. For elections to be credible, voters need to be convinced beyond doubt that what they see on the screen and what they press on the input device translates to their votes being properly counted by a system. If the code is black-boxed, how will voters ever know that their votes are counted properly? They wouldn't. Barriers to accountability would immediately rise. Paper-based election procedures today are designed with checks and balances so that the public (or its representatives that are a part of all stages of elections) can be assured that different parties cannot modify the results in any meaningful way. Even if deliberate or erroneous biases are introduced, such deviations would occur only on a local scale. The many eyes that watch the process ensure that any error or malicious vote-rigging does not happen on a national level. The problem with software-based systems is exactly their scalability. If someone found a way to manipulate the results in software voting machines that are spread around the country, the effect might be calamitous. Since most of the existing electronic voting systems come from a small group of commercial companies that want to protect their software under IPR laws [18], the existing situation under which unsafe black-boxed electronic voting systems are being used at the discretion of local officials is not acceptable. This transition to electronic voting cannot be justified, because any gains in efficiencies are counterbalanced by lost accountability. A recent report — based on analysis of Diebold's AccuVote closed source code — found that the system is far below even the most minimal security standards applicable in other contexts and is not suitable for general elections (Kohno, *et al.*, 2004). From this analysis, we learn not only of the dire situation of existing closed source electronic voting systems, but also of the importance of source access to the democratic process. As Kohno, *et al.* write:

"... an open process would result in more careful development, as more scientists, software engineers, political activists, and others who value their democracy would be paying attention to the quality of the software that is used for their elections. (Of course, open source would not solve all of the problems with electronic elections. It is still important to verify somehow that the binary program images running in the machine correspond to the source code and that the compilers used on the source code are non-malicious. However, open source is a good start.)" (Kohno, *et al.*, 2004)

The solution that seems to enable both accountability and modernization is to combine open source software with printed paper records that would turn electronic voting into a self-auditing system in which the many eyes of voters and auditors could vigilantly ensure correct results and uphold accountability. While open source introduces new complications to the system (its flexibility could be exploited in re-compilation), we would be better off with open source systems than with closed source environments. The physical audit trail helps guarantee accountability [19], and open sourcing the system guarantees scalability without fear of scaled errors (Neumann, 2002). Nobody is proposing that the voters will be

able to examine the source code as they vote. Instead, a reasonable arrangement would require voting machine manufacturers to make available their source code to scrutiny of government officials, members of political parties and other public entities. Such scrutiny would guarantee that software design and coding meets required standards. It would allow voters to better trust the system — not because they value one class of computer systems more than others — but because they could delegate their audit to those they trust, and not depend on the interests of private corporations.

To mention just one recent example where this approach has been taken, the Australian Capital Territory Electoral Commission has recently adopted this hybrid technology. On the selection of software they write: "The software for the electronic voting and counting system was built using Linux open source software, which was chosen specifically for this electoral system to ensure that election software is open and transparent and could be made available to scrutineers, [sic] candidates and other participants in the electoral process." (Australian Capital Territory Electoral Commission, 2003) To cite another example, VoteHere developed a software system for vote verification using an encrypted code in which voters receive receipts corresponding to their votes, which could be checked through the Internet to see that votes were tallied correctly. VoteHere recently released its source code for inspection (Zetter, 2004).

To summarize: the case of electronic voting shows that accountability can be achieved by involving many eyes in the process of software design and deployment. Most importantly, openness might not be a sufficient condition for accountability, but it most certainly is a necessary one.

Case II: Open source medical records

As in other sectors, a high level of accountability is a key prerequisite for the efficient operation of the health care and public health sectors. In a much more explicit way, government-mandated accountability opens a window into an attention-grabbing discussion. In 1996, "in order to combat waste, fraud, and abuse in health insurance and health care delivery" (U.S. Congress, 1996), Congress introduced the Health Insurance Portability And Accountability Act (HIPAA). HIPAA aims to achieve these goals

"by encouraging the development of a health information system through the establishment of standards and requirements for the electronic transmission of certain health information."

For the average health care provider or health plan, HIPAA requires activities, such as notifying patients about their privacy rights and how their information can be used, adopting and implementing privacy procedures; training employees so that they understand the privacy procedures; designating an individual to be responsible for seeing that the privacy procedures are adopted and followed; and, securing patient records containing individually identifiable health information so that they are not readily available to those who do not need them [20]. In other words, HIPAA mandates that health care providers and health insurers improve their accountability in theory and in practice to the immediate benefit of patients.

With the increased computerization of healthcare, a large portion of the effort to implement HIPAA was targeted to develop accountability in electronic systems. Of these, the most

vulnerable are those systems that deal with the electronic medical records (EMR) of patients. Indeed, the regulations imply that EMR software and data be protected within the confines of the healthcare provider or insurer, by, for example, a firewall system that prevents unauthorized access to systems and data. You might think that this is a straightforward requirement. However, a short investigation reveals a severe crisis if traditional software, operating under license agreements requiring "ownership without liability" is used. Consider, for example, the demands in a recent click-wrap license [21] that accompanied a recent update to Windows XP [22]. According to this license, in order to legally use Microsoft Windows on a continuous basis a user must accept software updates from time to time and at Microsoft's discretion while granting Microsoft (and its agents) limited access rights to hardware, software and data on a given machine. By electing not to accept updates and therefore preventing Microsoft access to a given computer and its files, a user voids her own license and will no longer be legally entitled to use Windows. Granting access might not be a problem in a less regulated environment, but when the programs that run on top of Windows are EMR applications, the demand to give the software vendor access to the data is in clear violation of HIPAA's requirement to limit access to patient related data [23]. This is simply one example of how accountability, even when mandated by law, is seriously put at risk by the "ownership without liability" regime which, by legal design, first and foremost ensures the well-being of software vendors and not the privacy of individuals.

HIPAA was enacted in order to prevent fraud and abuse and with the aim to elevate patients' trust in the healthcare system. With this in mind, should private software companies be included within the fragile circle of trust? What would it take for a healthcare provider to confide in a software company, and its unspecified "agents," in order to grant access to hardware, software, and patient data? Should healthcare providers simply accept a software vendor's promise to handle any and all data with care, or should they be able to closely monitor any software maintenance operations? What would happen to accountability if an attacker would find a bug in a program and use it in order to gain access to medical records? Considering common licensing terms that demand "ownership without liability" and click-wrap software licensing agreements that leave little room for negotiation, it is clear that users are forced to accept unfavorable terms. Hence it means that healthcare providers are forced to either compromise the privacy of patients, or violate binding commercial agreements. Accountability is lost in either case.

In a FLOSS environment, in contrast, the situation is different. In a fully operational EMR infrastructure available under an open source licensing agreement, the conflicting interests disappear and the "ownership without liability" impediment to accountability withers away. What is more, the same considerations that we encountered in the discussion on electronic voting systems apply here. An open system is inherently open to scrutiny. Not only can the operating system be monitored, but so can the applications that actually handle patient data. Open source licenses create a much more accountable software infrastructure for healthcare that upholds the requirements of HIPAA in a meaningful way. Indeed, free and open source offer true alternatives in the healthcare arena; two such projects are SPIRIT and Debian-Med. SPIRIT is a pioneering project partially funded by the European Commission that aims to accelerate the use of free and open source applications and information resources in the healthcare sector in Europe. SPIRIT distributes freely available software and medical information that enables better citizen-centered care (Spirit Project, 2003). Debian-Med is a version of Debian, a popular GNU/Linux distribution that aims

"... to develop Debian into an operating system that is particularly well fit for

the requirements for medical practice and research. The goal of Debian–Med is a complete system for all tasks in medical care which is build [sic] completely on free software." (Debian, 2003)

In sum, what we learn by looking at such projects is that promising first steps have been taken to ensure an adequate standard of care, effective quality assurance and debugging mechanisms, and reasonable licensing agreements that support accountability from the ground up [24].

The healthcare industry provides an interesting arena for investigating the "inevitable bugs" problem. An average personal medical record is comprised of dozens if not hundreds or thousands of details. Information coming from triage systems, lab reports, radiology, x–ray and imaging equipment, pharmacies, as well as hand–written observations and diagnoses from doctors, nurses and other experts needs to be combined in a secure way in one file. Naturally, the computerized systems that produce these ever–growing accumulations of data are manufactured by a multitude of firms, which more often than not compete with one another. When we consider the fact that the designers of each black–box software system have to develop their own model of the EMR world, the task of combining diverse data to a standardized, meaningful file seems daunting.

Given that "inevitable" bugs are usually those that arise due to inconsistencies in modeling, the diversity of computerized data sources in an average patient record is frightening. Annually nearly 195,000 people die in the U.S. as a result of potentially avoidable medical errors (Health Grades, 2004). It is well established that better-integrated EMR systems could potentially reduce this death toll significantly. However, despite ongoing efforts to develop standards of communication among medical devices [25] that would enable a fully standardized EMR system, conflicting interests of proprietary software vendors get in the way. These vendors perceive data streams produced by their devices as potential revenue streams to be exploited by offering complementary systems that use them, rather than key elements in a diversified patient record. Not only can bugs (Leveson, 1995a) be deemed inevitable in such an environment, but also accountability is lost. Each vendor could always accuse the others of not complying with standards, and no one can ever hope to have full scrutiny of the entire EMR system that relies on closed software.

In contrast, should the EMR data sources and management modules be all open–sourced, the likelihood of bug detection would increase by several orders of magnitude. Establishing deviations from agreed upon standards will be a trivial task and detecting errors in modeling will also be simplified. Most importantly, the EMR system could be tested as a system, and not as a haphazard amalgamation of black–boxed components. Needless to say, accountability in such an open–sourced environment will be well served. Vendors could not plead ignorance to their software's vulnerability since the code will be visible to all. In sum, opening the system up will ensure that patient records will cease to be hostages in battles between software vendors, falling prey to "inevitable" bugs that, in actuality, are preventable.



Further considerations to opening sources of accountability

Three potential objections

Objections to "open accountability" come from several directions. First, some FLOSS developers reduce their liability by including clauses in license agreements that stipulate that the software comes "as is," without any warranty, expressed or implied. What if an open source software related accident occurs, an objector might ask? How would we punish those responsible? Who would pay for damages? These are all good questions, but perhaps the wrong questions to ask.

We started this investigation with two assumptions:

1. increasing accountability grants value to society by motivating those most likely and able to prevent risk and harm to do so; and,
2. the existing status of accountability in software needs to be changed.

As we have seen, FLOSS advances all these aspects, and it does so better than the closed alternative, but, more importantly, in this process the very meaning of accountability is renegotiated. We lose "punishability" in its traditional sense and gain transparency. To analyze this we have to make two distinctions:

- (a) between systems developed by commercial entities and systems developed by volunteer developer communities; and,
- (b) between systems designed for corporate use versus systems designed for home users.

When firms are involved (either developers or users) liability can be arbitrated in court and it should be assigned on a case-by-case basis, and not as part of a sweeping strict liability principle. The use of open source methodologies in such cases would come to the aid of the court as it could be used to investigate the actions of actors involved. In the case of software systems developed by developer communities or for home users, the situation is different. Here, except in extreme cases:

- (a) most probably the transaction costs of arbitration are usually too high to bring the case before the courts (although class action is sometimes an option); and,
- (b) it is not clear what the legal definition of a developer community is (thus undermining the motivation for court intervention).

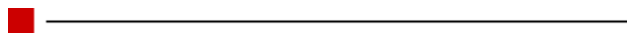
In consequence, it is unlikely to expect a legal solution to liability problems when non-corporate users are involved. Instead, however, the solution is to be found in the social realm. We just have to part with the idea of strict liability in those cases, because, as I have argued, to maximum benefit will be achieved through maintaining a balanced solution. In addition, FLOSS offers a higher degree of control — and therefore, also of self-responsibility, which, on average, reduces errors. In conclusion, you should be convinced by now that accountability, in its modified, meaning, will be fortified and that the overall benefit to society will increase.

A second objection might ask whether FLOSS can actually reach the necessary performance levels required in mission-critical applications, and whether some degree of closure is not required in order to ensure adequate performance. To answer the first part of this question it is enough for us to look around the software world today. Dozens of examples from

everyday practice illustrate FLOSS projects being deployed in healthcare, defense, government, research and all other fields where high performance, secure, sometimes life-critical software is used around the world and beyond [26]. In many cases these open systems replace older, proprietary, systems that become obsolete exactly because of their closed nature. But can accountability be maintained in an open environment when conflicting values like utmost performance, personal privacy, or national security enter the equation? In exactly these situations in which competing values play a significant role, the openness of the system guarantees that any of the competing values will be compromised.

In an open environment, software is open to inspection. As in the case of a constitutionally defended free and open press that safeguards the operation of government, and as in the case of elite restaurants that keep their kitchens open for all to see, so can FLOSS promote the proper operation of computerized systems and mission critical applications. Naturally, not all situations can benefit from all aspects of the FLOSS framework [27]. Yet taken as a whole, it is evident that in exactly the mission critical areas system openness is a safeguard not only to accountability but also to improved performance in the long run. FLOSS reduces the overall price that society pays by ensuring that people learn from mistakes and by removing commercial interests from the equation.

A third line of objection would claim that FLOSS are nothing more than a short-lived fad, and surely not a panacea to the many problems facing our computerized society. This view focuses on development methodology and obscures a more important discussion on competing values in a society of technophiles that constantly gravitates towards increased levels of automation, unaware of the great costs to its social fabric. I would argue that, in the case of FLOSS, the method is the message. FLOSS started not as technological solutions to technological problems but rather as social solutions to a techno-legal Gordian knot of impossible licensing agreements [28] that betrayed the beliefs of the free software movement founders. Since then, these systems have developed to become sustainable alternatives to traditional production systems of firms and markets (Benkler, 2002). This shift in the control of the means of production promotes a discussion of the social implications of technology. The engineering community has proven, through the practice of FLOSS, that its diversity protects important values, accountability notwithstanding [29]. As we have clearly seen, the methodologies of FLOSS are not at a technological fringe, but rather at the center. Employing them more widely will enable us to find the right balance between social values and technological innovation.



Conclusion

The founders of the free and open source movement, and their scores of followers, have provided many examples of how accountability can be maintained despite the barriers and challenges presented by our computerized society. By ensuring that software's source code remains free through sensible licensing agreements, by guaranteeing that enough eyes watch the many hands that fix bugs, and by accommodating new modes of collaborative activity through social — and not legal — mechanisms, we can sincerely hope that the barriers to accountability will diminish. At the same time, we have a responsibility to build a framework for moral and ethical deliberations that can accommodate meaningful discussions regarding rapidly changing technological practices. By contrasting FLOSS with closed systems, we are asked to rethink the importance of accountability and dissect some of its conflicting meanings as we adopt accountability to the ever changing world of

information technology. 

About the author

Shay David is a doctoral student in Science and Technology Studies and is affiliated with the Information Science Program at Cornell University in Ithaca, N.Y. With several years of hands-on software development experience, his research now focuses on the cultural, social and political aspects of collaborative modes of knowledge production in general and FLOSS particularly.

Web: <http://www.shaydavid.info>

E-mail: sd256 [at] cornell [dot] edu.

Acknowledgements

I am highly indebted to the friends and colleagues that have commented on early drafts of this paper, specifically, Helen Nissenbaum, Tarleton Gillespie, Michal Tsur and the participants of the FLOSS panel at the 4S 2004 annual meeting. In the spirit of FLOSS I have benefited from an open exchange of ideas with these pundits, however, any errors remain my own.

Notes

1. Understanding this co-production is a key subject matter in the social study of science and technology. An elaborate set of essays developing this theme is available in Jasanoff (2004).
2. For a more detailed accounts on the social history and historical development of the free software and open source movements see David (2003); Weber (2004); DiBona, *et al.* (1999); and, Stallman and Rubin (1987).
3. See, for instance, Johnson and Mulvey (1995).
4. See for example a detailed discussion on the successful use of open source systems by NASA; Norris (2003).
5. As Nissenbaum rightly identifies when reading engineers like Parnas, *et al.* (1990); and, Corbató (1991). See also more recent accounts that advocate this view as reported in Neumann's report on risks to the public in computerized systems; Neumann (2002).
6. These types of bugs are often the Achilles heel of many programs which malicious viruses easily attack.
7. It has been estimated that 50–60 percent of the software development effort goes into

testing. However, more testing should be accepted by the industry (Osterweil, 1996).

8. To name just one recent example, recall the excitement around Total Quality Management (TQM), the panacea of the late '80s and early '90s that has been avidly adopted by software mavens. See, for example, Zultner (1993). TQM has since been overridden by "better" quality assurance techniques that, just like TQM, over-promise and under-deliver.

9. As defined by Feinberg (1985).

10. In contrast to Nissenbaum (1996) and to Vedder (2001) that discuss the philosophical groundings of liability and the challenges when transporting it from law to ethics, but still conclude that the strict liability is appropriate in some situations.

11. Perhaps the most seminal example of this process in action is the Apache Web server, which started out as a "patchy" piece of software (hence Apache), but over the years has evolved to overcome "environmental" problems introduced by the dynamics of the software arena and the ever-changing variations in so called standard protocols. See Behlendorf (1999). Recognizing how stable Apache has become, it was adopted by most Web developers and is today the most widely used Web server on the Internet, driving over 70 percent of all Web pages. See Web Server Survey (2004).

12. On objections to this argument see the [conclusion](#) to this paper.

13. Specifically see Locke's second treaty which discusses the "right to possessions."

14. GNU is a free operating system which is distributed today as part of the GNU/Linux package.

15. The GPL has been the site of many debates within the free software/open source community itself, and the main cause for a split in the community between free software devotees and open source supporters who created other, less rigid, forms of open source licensing agreements. Essentially, when compared with traditional software licensing agreements, many of the free software/open source agreements maintain a much attenuated version of "ownership."

16. How many of us have sent a wrong e-mail message to the wrong person only because of some "smart" auto-complete feature, which we did not invoke, "helped" us send it?

17. As an anecdote that shows how trouble in these systems can come from unexpected sources, consider the following story: "The *Bangkok Post* (27 March 2002) reports that, following voting in an election on 3 March 2002, mice managed to climb into one of the ballot boxes and chew up ballots. The winning candidate had a 65-vote lead when the undamaged votes were counted, and it was estimated that scraps left by the mice represented another 40 ballots. 'This result still has to be endorsed by the Election Commission,' reports The Post." (Neumann, 2002).

18. Specifically, Diebold Inc. and Sequoia Voting Systems dominate the market.

19. Objectors to a paper trail cite "improper voter influence" and laws that forbid the

removal of anything from polling sites as reasons for their objection, but technical solutions to these problems have been proposed by Chaum and others (Chaum, 2003).

20. U.S. Department of Health and Human Services, 2003. "HIPAA Frequently Asked Questions," at http://answers.hhs.gov/cgi-bin/hhs.cfg/php/enduser/std_alp.php?p_sid=xPmDQ6Gg&p_lva=&p_li=&p_page=1&p_cat_lv11=7&p_cat_lv12=%7Eany%7E&p_search_text=&p_new_search=1.

21. Click-wrap licenses are those that require the user to click "I agree" in order to install software. These licenses often include ridiculous or even illegal clauses. Example of click-wrap licenses and their absurdity can be found at Ress (2004).

22. See the detailed discussion of Microsoft Windows XP SP1 End-user Licensing Agreement and its violation of HIPAA at slashdot.org, at <http://ask.slashdot.org/article.pl?sid=02/08/27/2030205&mode=thread&tid=109>.

23. One can find legal and technical arguments to show that in this specific case there is a workaround that evades this fatal conclusion. For instance, the user can disconnect herself from the external network altogether and claim not to know that updates are available, or conversely, the updates could be handled offline using only trusted media, or furthermore, someone can claim that updates are necessary to the operation of the system. Even so, this example clearly shows how accountability's erosion is deepened by draconian proprietary software licensing regimes.

24. For a detailed account of the politics of open source adoption in healthcare, see David (forthcoming).

25. Most notably a protocol called Health Level 7 (HL7).

26. The latest Mars Lander spacecraft, the European Beagle (named after Darwin's ship to denote the historical importance of the journey), was guided to Mars by a Linux-based flight control system. See Williams (2003). As it happened, the Beagle crashed on the surface of Mars, but it is exactly in the case of accidents that the openness of the system becomes important.

27. We'd be hard pressed to imagine that a space mission can enjoy the luxury of "test-use cycles" and a "release-often release-early" philosophy.

28. For a detailed analysis of the history and social construction of free and open source software, see David (2003).

29. See, for example, Eriksén (2002) for an engineering perspective of the importance of accountability in human-computer interaction system design.

References

Australian Capital Territory Electoral Commission, 2003. "Electronic voting and counting," at <http://www.elections.act.gov.au/Elevote.html>, accessed 16 December 2003.

- Brian Behlendorf, 1999. "Open source as a business strategy," In: Chris DiBona, Sam Ockman, and Mark Stone (editors), 1999. *Open sources: Voices from the open source revolution*. Sebastopol, Calif.: O'Reilly & Associates.
- Yochai Benkler, 2002. "Coase's penguin, or Linux and the nature of the firm," *Yale Law Journal*, volume 112.
- Jorge Luis Borges, 1998. "A universal history of infamy," In: *Collected fictions*. New York: Viking Penguin.
- Michel Callon, 1998. *The laws of the markets*. Oxford: Blackwell.
- Miguel de Cervantes, 2001. *Don Quixote*. Translated by T. Smollett. Volume I. New York: Modern Library.
- David Chaum, 2003. "Secret-ballot receipts and transparent integrity," at <http://www.vreceipt.com/article.pdf>, accessed 16 December 2003.
- F.J. Corbató, 1991. "On building systems that will fail," *Communications of the ACM*, volume 34, number 9, pp. 73–81.
- Shay David, forthcoming. "Healthcare, a F/OSS opportunity," In: J. Karaganis and R. Latham (editors). *The politics of open source adoption*.
- Shay David, 2003. "On the uncertainty principle and social constructivism: The case of free and open source software," at <http://www.shaydavid.info/papers/uncertainty.pdf>.
- R.M. Davison, 2000. "Professional ethics in information systems: A personal perspective," *Communications of AIS*, volume 3, number 8, pp. 1–34.
- Debian, 2003. "Debian Med," at <http://www.debian.org/devel/debian-med/>, accessed 16 December 2003.
- Chris DiBona, Sam Ockman, and Mark Stone (editors), 1999. *Open sources: Voices from the open source revolution*. Sebastopol, Calif.: O'Reilly & Associates.
- Nicolas Ducheneaut, 2003. "The reproduction of open source software programming communities," PhD dissertation, Information Management and Systems, University of California, Berkeley.
- Sara Eriksén, 2002. "Designing for accountability," Paper read at Second Nordic Conference on Human–computer Interaction, Aarhus, Denmark.
- J. Feinberg, 1985. "Sua Culpa," In: D.G. Johnson and J. Snapper (editors). *Ethical issues in the use of computers*. Belmont, Calif.: Wadsworth.
- Health Grades, Inc., 2004. "Patient safety in American hospitals," at http://www.healthgrades.com/media/english/pdf/HG_Patient_Safety_Study_Final.pdf, accessed 8 November 2004.

Thomas Hobbes, 1651. *Leviathan, or, The matter, forme, and power of a common wealth, ecclesiasticall and civil*. London: Andrew Crooke.

Perry Hoberman, 2003. "Accept," at <http://www.perryhoberman.com/accept/>, accessed 8 November 2004.

Sheila Jasanoff (editor), 2004. *States of knowledge: The co-production of science and social order*. London: Routledge.

Deborah G. Johnson, 2000. *Computer ethics*. Third edition. Englewood Cliffs, N.J.: Prentice-Hall.

Deborah G. Johnson and John M. Mulvey, 1995. "Accountability and computer decision systems," *Communications of the ACM*, volume 38, number 2, pp. 58–64.

Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach, 2004. "Analysis of an electronic voting system," Paper read at IEEE Symposium on Security and Privacy 2004 (27 February).

Bruno Latour, 1987. *Science in action: How to follow scientists and engineers through society*. Cambridge, Mass.: Harvard University Press.

Nancy G. Leveson, 1995a. "Safety as a system property," *Communications of the ACM*, volume 38, number 11, p. 146.

Nancy G. Leveson, 1995b. *Safeware: System safety and computers*. Reading, Mass.: Addison-Wesley.

John Locke, 1821. *Two treatises on government*. London: R. Butler.

Eben Moglen, 1999. "Anarchism triumphant: Free software and the death of copyright," *First Monday*, volume 4, number 8 (August), at http://www.firstmonday.org/issues/issue4_8/moglen/.

James H. Moore, 1985. "What is computer ethics?" In: T.W. Bynum (editor). *Computers and ethics*. Oxford: Blackwell.

Peter G. Neumann, 2002. "Risks to the public in computers and related systems," *ACM SIGSOFT Software Engineering Notes Archive*, volume 27, number 3, pp. 5–19.

Helen Nissenbaum, 1996. "Accountability in a computerized society," *Science and Engineering Ethics*, volume 2, number 2, pp. 25–42.

Helen Nissenbaum, 1995. "Should I copy my neighbor's software?" In: D.G. Johnson and H. Nissenbaum (editors). *Computers, ethics, and social values*. Englewood Cliffs, N.J.: Prentice-Hall.

Jeffrey S. Norris, 2004. "Mission-critical development with Open Source software: Lessons learned," *IEEE Software*, at <http://www.computer.org/software/homepage/2004/s1nor1.htm>, accessed 23 August 2004.

Open Source Initiative, 2004. "Open Source Initiative," at <http://www.opensource.org/index.html>, accessed 4 May 2004.

Leon Osterweil, 1996. "Strategic directions in software quality" *ACM Computing Surveys (CSUR) Archive*, volume 28, number 4, pp. 738–750.

D. Parnas, J. Schouwen, and S.P. Kwan, 1990. "Evaluation of safety–critical software," *Communications of the ACM*, volume 33, number 6, pp. 636–648.

Eric S. Raymond, 1999. *The cathedral and the bazaar: Musings on Linux and open source by an accidental revolutionary*. Sebastopol, Calif.: O'Reilly & Associates, and at http://www.firstmonday.org/issues/issue3_3/raymond/.

Manon Ress, 2004. "CPTech's Survey of Current Shrinkwrap and Clickwrap Contracts," Consumer Project on Technology, at <http://www.cptech.org/ecom/ucita/licenses/>, accessed 23 August 2004.

slashdot.org, 2002. "Is Win2k + SP3 HIPAA compliant?" at <http://ask.slashdot.org/article.pl?sid=02/08/27/2030205&mode=thread&tid=109>, accessed 16 December 2003.

Brian Cantwell Smith, 1985. "The limits of correctness," *ACM SIGCAS Computers and Society*, volumes 14–15, number 1, pp. 18–26.

Spirit Project, 2003. "Spirit Project 2003," at <http://www.euspirit.org/en/project.php>, accessed 16 December 2003.

Richard M. Stallman, 2002. "Why software should have no owners.," In: *Free software, free society: Selected essays of Richard M. Stallman*. Cambridge Mass.: Free Software Foundation.

Richard M. Stallman, 1987. "The GNU Manifesto," In: *GNU Emacs manual*. Cambridge, Mass.: Free Software Foundation.

Richard M. Stallman and Paul Rubin. 1987. "GNU Project — Free Software Foundation," *GNU's Bulletin*, volume 1, number 3 (19 June), at <http://www.gnu.org/bulletins/bull3.html>, accessed 4 May 2004.

U.S. Congress, 1996. "Health Insurance Portability And Accountability Act Of 1996 (HIPAA)," (21 August).

U.S. Department of Health and Human Services, 2003. "HIPAA frequently asked questions," at http://answers.hhs.gov/cgi-bin/hhs.cfg/php/enduser/std_alp.php?p_sid=xPmDQ6Gg&p_lva=&p_li=&p_page=1&p_cat_lv11=7&p_cat_lv12=%7Eany%7E&p_search_text=&p_new_search=1, accessed 16 December 2003.

Anton Vedder, 2001. "Accountability of Internet access and service providers: Strict liability entering ethics?" *Ethics and Information Technology*, volume 3, number 1, pp. 67–74.

Web Server Survey 2004. "NetCraft 2004," at <http://news.netcraft.com/>, accessed 4 May 2004.

Steve Weber, 2004. *The success of open source*. Cambridge, Mass.: Harvard University Press.

Peter Williams, 2003. "Linux set for Mars landing," National Space Centre (Leicester), at <http://www.vnunet.com/News/1151517>, accessed 16 December 2003.

Kim Zetter, 2004. "See-through voting software," *Wired News* (8 April).

Richard E. Zultner, 1993. "TQM for technical teams," *Communications of the ACM*, volume 36, number 10, pp. 79–91.

Editorial history

Paper received 22 September 2004; accepted 19 October 2004.

[Contents](#) [Index](#)

Copyright ©2004, *First Monday*

Copyright ©2004, Shay David

Opening the sources of accountability by Shay David
First Monday, volume 9, number 11 (November 2004),
URL: http://firstmonday.org/issues/issue9_11/david/index.html