NBER WORKING PAPER SERIES

HALF A CENTURY OF PUBLIC SOFTWARE INSTITUTIONS: OPEN SOURCE AS A SOLUTION TO HOLD-UP PROBLEM

Michael Schwarz
Yuri Takhteyev

Working Paper 14946
http://www.nber.org/papers/w14946

NATIONAL BUREAU OF ECONOMIC RESEARCH
1050 Massachusetts Avenue
Cambridge, MA 02138
May 2009

We are grateful to Jacques Crémer, Ara Keys, Paul Laskowski, Luisa Schwartzman, Marshall Van Alstyne and Rajesh Veeraraghavan for comments and suggestions. We also would like to thank for comments seminar participants at CMU, Microsoft Research and Fifth bi-annual conference on the Economics of the Software and Internet Industries. The views expressed herein are those of the author(s) and do not necessarily reflect the views of the National Bureau of Economic Research.

© 2009 by Michael Schwarz and Yuri Takhteyev. All rights reserved. Short sections of text, not to exceed two paragraphs, may be quoted without explicit permission provided that full credit, including © notice, is given to the source.

Half a Century of Public Software Institutions: Open Source as a Solution to Hold-Up Problem
Michael Schwarz and Yuri Takhteyev
NBER Working Paper No. 14946
May 2009
JEL No. D45,D62,D64,H4,H44,L17,L3,N8,O3,O31,O43

ABSTRACT

We argue that the intrinsic inefficiency of proprietary software has historically created a space for alternative institutions that provide software as a public good. We discuss several sources of such inefficiency, focusing on one that has not been described in the literature: the underinvestment due to fear of holdup. An inefficient holdup occurs when a user of software must make complementary investments, when the return on such investments depends on future cooperation of the software vendor, and when contracting about a future relationship with the software vendor is not feasible. We also consider how the nature of the production function of software makes software cheaper to develop when the code is open to the end users. Our framework explains why open source dominates certain sectors of the software industry (e.g., the top ten programming languages all have an open source implementation), while being almost none existent in some other sectors (none of the top ten computer games are open source). We then use our discussion of efficiency to examine the history of institutions for provision of public software from the early collaborative projects of the 1950s to the modern "open source" software institutions. We look at how such institutions have created a sustainable coalition for provision of software as a public good by organizing diverse individual incentives, both altruistic and profit-seeking, providing open source products of tremendous commercial importance, which have come to dominate certain segments of the software industry.

Michael Schwarz
Yahoo! Research
2397 Shattuck Ave
Berkeley, CA 94704
and NBER
mschwarz@yahoo-inc.com

Yuri Takhteyev
UC Berkeley
102 South Hall
Berkeley CA 94720-4600
yuri@sims.berkeley.edu

1. Introduction

Nonrival creative works, such as books, music and films, are subject to copyright protection, which makes such works excludable and allows their producers to sell copies for profit, strengthening incentives for private provision of such goods. Since 1980 the copyright law has offered the same protection to producers of software. However, while successful artists and publishers nearly always take full advantage of copyright protection, many successful software engineers and technology companies choose not only to allow others to use their software without paying royalties, but also to modify and redistribute it with almost no restrictions. Such public software, known since the late 1990s as "open source," has grown to a staggering quantity, serving such a wide range of purposes that it is possible for many people to rely on such software for all of their personal computing needs.[1] By contrast, consumers of books, music and films can hardly satisfy their needs exclusively through content that allows unrestricted reproduction.[2]

The academic discussion of open source has primarily focused on understanding the incentives that may lead individual actors to contribute resources to the production of such software. Johnson (2002) considers the incentives of self-interested agents to contribute to open source and investigates the effects of changes in the population size of user programmers on the provision of open source.[3] Athey and Ellison (2006) consider a model where programmers have altruistic preferences and derive different allocations of efforts.[4] Lerner and Tirole (2002) point out that contributors to open source derive gratification from peer recognition and may increase

1 One of the authors has relied on such software almost exclusively for the last several years, making use of a free operating system, free office suite, free software development tools, free image processing equipment, and free statistical software, among many other applications. The only exception has been the proprietary plugin for viewing multimedia content in Adobe Flash format.

2 Wikipedia provides a rare instance of a highly popular written work distributed with a license that allows unrestricted copying and mostly unrestricted modification. (In particular, this license allows any company or individual to download Wikipedia's content *en masse* and to setup an alternative website displaying this content, as long as the content is offered with the original license and acknowledges the authors.) Notably, Wikipedia's license was originally introduced for software manuals, as an adaptation of a license used for distribution of public software such as Linux.

3 There is an extensive literature investigating mechanisms for the provision of public good by private actors, e.g., Bliss and Nalebuff (1984), Bergstrom, Blume, and Varian (1986), Bagnoli and Lipman (1989), Varian (1994).

4 Lerner, Pathak and Tirole (2006) investigate the connection between the incentive to contribute to an open source project and the project size and perceived likelihood of success.

their future earnings by producing works that can show their skills to employers. Surveys of contributors have found a range of motivations, the simplest of which can be summed up with the phrase “Just for Fun” — the title of the autobiography of the creator of Linux, the most widely used open source operating system (Torvalds 2001).[5] Altruism, career concerns, and intrinsic motivations are essential drivers of open source movements. However, the same motives can and do exist in many other industries, without producing the same effect.[6]

The fact that people and firms contribute to public software is therefore not a puzzle *per se*. What requires explanation is the success of such software in competition with proprietary software produced by profit-motivated entities. Why can open source software, created by the efforts of part-time volunteers and weakly incentivized contributors, successfully compete with commercial software created by private enterprise?[7] While open source software has the obvious advantage of being free, price cannot explain its disproportionate popularity among highly sophisticated users (both individuals and businesses) who are likely to value quality and performance at least as much as other users. Individual users of Linux, for example, often use it as a replacement for a proprietary operating system that they have already paid for when purchasing a computer. However, the use of open source by individuals is only the tip of the iceberg compared to its use by corporations. Internet companies such as Amazon, Facebook, Google, Yahoo! and many others use open source operating systems and tools for some of their most challenging computing tasks. The world’s largest supercomputer (IBM Roadrunner) and the world’s largest database (Yahoo!) are both based on open source software (IBM 2008, Lai 2008).

Understanding public software by focusing on individual incentives is also problematic because such incentives are diverse, vary between projects and participants, and change over

5 Lakhani and Wolf (2005) report that 44.9% of their respondents say that they contribute to open source projects because the work they do for the project is “intellectually stimulating.” Hars and Ou (2001) find similar motivation (“desire to code”) for 79.7% of the responders. Other common motivations involve the opportunity to learn (dominant in Ghosh 2005), the need for the software (dominant in Lakhani and Wolf 2005), desire to share with others, and many other.

6 Lerner and Tirole (2002) suggest that open source developers can use their participation in open source to signal their talent to employers. Note again that while aspiring writers similarly need to demonstrate their ability to the publishers, this need has not led to proliferation of free literature.

7 A similar question is asked by Weber (2005), who approaches it from the organizational perspective, looking to understand how open source development is coordinated.

time. Such change becomes most apparent when we extend our discussion from modern "open source" institutions that have taken shape over the last ten years to the different institutions that have provided software as a public good since the early 1950s. For example, large corporations are today among the most important contributors to public software. By one estimate, employees of just five companies (Red Hat, IBM, Novell, Intel and Oracle) jointly contributed 32% of the changes for a recent release of the Linux kernel, while the contributions of individual volunteers likely added up to less than that (Corbet 2007). Such contributions are made for reasons that are quite different from those that motivated individual programmers' contributions to some of the same projects in the 1980s, AT&T's and the US Government's contributions in the 1970s, or IBM's contributions in the 1950s and 1960s.

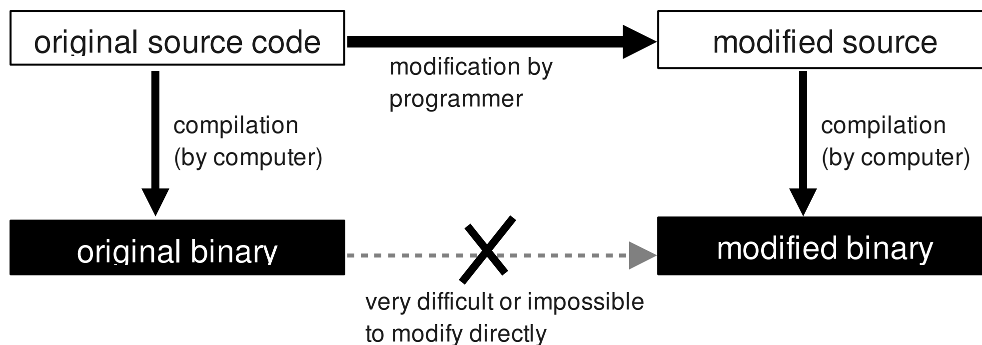
For this reason, instead of focusing on individual incentives *per se*, we start with a look at the inefficiencies inherent in the provision of software as an excludable good, stressing that such inefficiencies arise not just because of deadweight loss due to monopoly pricing, but also due to the hold-up problem and other inefficiencies intrinsic to the production of proprietary software.

We identify a source of inefficiency associated with proprietary software that has not been described in the earlier literature: proprietary software causes underinvestment in complementary products and technologies due to the fear of hold up. Users of proprietary software must rely on the original vendor for changes to the software and may be reluctant to make complementary investments in anticipation of future hold-up when contracting about future relationship with the software vendor is not feasible (e.g. due to uncertainty about the type of modifications that original software will need in the future). Such underinvestment can be especially large if the user has unusual requirements and expects to need highly customized modifications. This means, that hold up presents a particularly serious problem for the most sophisticated users of software. Hold up explains why open source software dominates some sectors of the industry, while playing negligible role in others. For example, open source software is highly present and often dominant in software development platforms (operating systems, programming languages, libraries and databases), while having negligible market share among such applications as computer games.

We then look at how the intrinsic inefficiency of proprietary software has historically created a space for alternative institutions that provide software as a public good, illustrating this process with historical examples.

2. Efficiency and Excludability in Software

Software can exist in two very different forms, best understood as distinct goods that differ both in their utility and excludability. Software is created by writing instructions in the form of a highly formalized but human-readable text. This representation of the program, called "the source code," can then be automatically converted into a *binary* form that can be executed by the computer. Binary software can be *used*, but cannot be easily *modified*. To change the behavior of the software, one needs to modify its source code and then produce a new binary (see Figure 1). Access to the source code can thus provide the user with a substantial additional value: the ability to correct the software, improve it, customize it or extend it to new situations.



We start this section by explaining why distributing software in the form of the source code may often be more efficient than distributing binary software. We identify several reasons for such gains: one that we believe to be novel (the hold-up problem inherent in trading binary software), as well as several other reasons noted in the earlier literature. We then proceed to explain why distributing software in the form of source code often renders it non-excludable.

2.1. The Hold-Up Problem in Binary Software Transactions

When users purchase binary software, they must rely on the original vendor to supply them with any modifications that may later prove to be necessary for their continued use of the software. For example, a user of payroll software may face a need for modifications reflecting changes in accounting rules, while a user of a web server may discover a need for changes to protect the server against a recently discovered intrusion method. The exact nature and cost of such future modifications often cannot be foreseen *ex ante*. Their price and quality must therefore be negotiated *ex post.*[8]

This poses a problem for users who need to make investments complimentary to the software. For example, the user of payroll software may have to invest in training, while the user of the web server may have to purchase compatible hardware and develop additional software to link the web server with its business processes. The user's ability to recoup the value of such investments may depend on the software vendor's cooperation, since only the original vendor has the source code that is needed for producing a modified version. This makes such complementary investments relationship-specific. Use of binary software may therefore create a hold-up problem: the vendor gains bargaining power and the ability to negotiate a larger share of the profit *ex post*. Such risk of hold-up may lead the potential user to either forgo the purchase of the software or to reduce complementary investments. [9]

The risk of hold-up is particularly severe if the complementary investments are large or when the user has unusual requirements that would make the need for custom modifications more likely. The problem would also be amplified if the producer and the user are competing in

8 There is usually substantial ambiguity between what constitutes a defect and what is an idea for a future improvement. Even in the case of defects, the vendor cannot commit to fixing *all* future defects, since the cost of doing so cannot be known before hand. Software vendors consequently disclaim warranties to the maximum extend allowed by law. For example, the end user license agreement for Windows Vista disclaims all warranties beyond the promise that "the software will perform substantially as described in the Microsoft materials" the user receives with the software. The license terms further limit Microsoft's reponsibility to a refund of the original price. (In this sense, Windows and Linux come with identical warranties: the user can never hope to recover more than the price they paid for the software.)

9 For a general discussion of inefficiencies due to hold up see Williamson (1975, 1979), Klein, Crawford and Alchian (1978) and Hart and Moore (1990).

other markets. Such reasons would lead us to expect companies like Google or Yahoo! to avoid purchasing binary software, especially when it is supplied by current or potential competitors and is intended for use in core infrastructure, where their needs are likely to be unique and where cooperation of the producer will be crucial for continued functioning of the business.

Vertical integration is a standard "second best" solution to the hold-up problem. In case of software, vertical integration takes the form of in-house production of software by the user. The non-rival nature of software, however, introduces an important difference from the more traditional hold-up scenarios: users who develop software for their own needs may incur little or no cost from offering it to others. Selling binary version of the software to other potential users may be impossible, since the other users may be just as reluctant to use software provided to them without the source code. The user who produces software for their own reasons may therefore choose to share the source code. As we explain below, sharing the source code with a permission to modify it may make it impossible to charge a positive price for it. However, benefits from network externalities can make sharing a preferred strategy.

2.2. Additional Efficiency Gains

In addition to reducing underinvestment due to fear of hold-up, software distributed with the source code often has lower total social cost of development. Some of those efficiency gains have been discussed in earlier literature and we consider them here only briefly, even though, as we will see, some of those reasons have been historically quite important for the provision of public software.

Distribution of source code can lead to efficiency gains by making it possible for the modifications to be done by those actors who have the best information about their value are are best equipped to carry them out. While the original producer of the software may sometimes be able to improve or customize it most efficiently (e.g., by virtue of already having deep knowledge of the software), this is not always the case. Quite often the users may be in a better position to make the improvements. The limitations of the software are often discovered naturally by the

users, while the producer may have to undertake prolonged testing, as errors may occur with low frequency or only when the software is used in specific ways that the programmers could not anticipate.[10] In other words, to improve the software, the producer must acquire users' knowledge of how the software should behave and of how it actually behaves in practice. Such process of knowledge transfer can be costly (von Hippel 1998, 2007). Letting users change the software and share the modifications among themselves may often lead to a substantial reduction of the total social cost in software production.

The cost of finding errors may also be higher for the original producer because of a principal-agent problem investigated by Johnson (2006): a firm that withholds the source code from the users must rely on its own programmers to find the errors in the code, yet career concerns may lead the programmers to collude and avoid revealing their own and each others' errors. Johnson's principal-agent argument can be extended to point out that the firm may also forgo opportunities for efficient improvements because career concerns may lead the firms' agents to deny the existence of the problems identified by the users.

Finally, trading binary software and withholding its source code increases the overall social cost of software development, because it prevents producers from fully re-using existing solutions developed by other firms. While software developers can license binary software modules and incorporate them into their own products, this creates a hold-up problem similar to the one described earlier and may involve substantial transaction costs incurred while the legal and strategic implications of the transaction are analyzed.

2.3. Excludability

Despite the losses in aggregate efficiency outlined above, software is typically sold in binary form. The reason for this is the higher excludability of binary software. While binary software is non-rival and can be replicated at negligible cost, the law renders it effectively excludable by making it illegal to copy it without the producer's authorization. The producer can

10 A variation of this argument was introduced by Raymond in 1997, later published in Raymond (1999/2001).

further augment this excludability by adding technical protections against unauthorized copying, for instance by requiring the users to enter unique authorization codes to unlock the software.

Source code, by contrast, is not as easily excludable. While the same legal protections apply in theory, in practice, licensing software in the source code form is usually not a viable option. First of all, access to the source code allows the user to remove any technical restrictions on unauthorized copying or use. Second, and more importantly, the fact that source code can be modified creates serious problems for establishing its relationship to the original software. When someone makes an unauthorized copy of binary software, they obtain an *exact copy,* which can later be easily shown to be "the same" as the original software. Access to the source code, on the other hand, makes it possible to copy elements of the software into another product in ways that would be difficult to prove. Additionally, the recipient can use the source code as a blue-print to develop an alternative product at a fraction of the original cost but without provable violation of the law.

For those reasons, contracts that give one party access to another party's source code while limiting how such source could be used are inherently hard to enforce and expose both parties to the risk of expensive litigation. This risk is best illustrated by the $3 billion lawsuit brought against IBM by SCO over IBM's alleged misuse of Unix source code, which cost each party millions of dollars in legal fees. While such licensing agreements are used occasionally, transaction costs limit their use.[11] Thus, as a first-order approximation, one can think of software in binary form as an excludable good and software in source code form as non-excludable after the first sale.[12]

11 For example, prior to the United States v. Microsoft case trial (1998-2001), Microsoft shared its Windows code with no other organization. (It later started making Windows source code available to select clients in response to the anti-trust charges brought by EU and to the demand of users who threatened to switch to open source.)

12 In addition to the high transaction costs, licensing source code with restrictions may fail to fully resolve the hold up problem. For example, the user may not have the technical capacity to make the necessary modifications, and would need the right to hire a third party to do this work, transferring the code to them. The companies with the best capacity to offer such service may well be competitors of the original vendor. The users may also need the ability to share the modifications, since otherwise they may lack the necessary economies of scale and face prohibitive costs of modifications. Solving the hold-up problem thus requires making the software non-excludable upon the first sale.

2.4. Institutional Solutions

As we have argued, the total social cost of software production may often be minimized if software producers allow any party to modify their software and redistribute the modifications. The hold-up problem largely disappears, allowing users to make optimal investments into complementary goods. Improvements to the software can be made by users as they discover the need in such improvements. Software producers can reuse all existing solutions, including those that anticipate their needs only partly.

Allowing unlimited replication of the software, however, reduces the producers' ability to charge for it, leading to underinvestment due to free-rider problems. For this reason, proprietary software distributed in binary form may emerge as the dominant form of provision, despite the inefficiencies mentioned above. We would especially expect this to be the case for software that does not require large complementary investments, is unlikely to need modifications, and is offered to users that have no capacity to modify software even when the source code is offered to them. Computer games offer a quintessential example of such software: they are typically used for a limited period of time, rarely require substantial game-specific complementary investments, offer limited opportunities for useful modifications, and are mostly offered to consumers with no programming skills. The share of public software in computer games is in fact negligible.

On the other hand, for software products that require substantial complementary investments and where the users have the capacity and frequently the need to modify source code, the inefficiencies inherent in excludable software become a substantial factor. In such cases, the actors could reap substantial benefits if they could form a coalition to provide software as a public good. The emergence of such coalitions would be facilitated when a number of potential beneficiaries is so small that they can literally sit at the same table and agree on how to share the cost. As we will see, such smaller coalitions have in fact formed repeatedly from the earliest days of software development.

Such coalitions may become harder to form or maintain when the benefits are spread among a larger number of individuals. Early coalitions, however, may over time evolve into more

sophisticated institutions that organize individual incentives for contributing and a reduce transaction costs of cooperation. Institutions can be understood as “the humanly devised constraints that shape human interactions” (North 1990, p. 3). They can be formal or informal and can affect the behavior of the actors by changing either actors’ choice sets or their preferences. Effective institutions make possible certain forms of cooperation that would be impossible without them. Such effective institutions developed over time, as the actors look for ways to improve the framework within which their interaction takes place. For example, economic historians have pointed out that the efficiency of modern markets is made possible by the existence of a legal framework that have developed over the centuries (e.g., North 1981).

In the rest of this paper we present the history of the institutions that have provided public software over the last half century, from the early coalitions to the current institutional forms, which often make it possible to bring together hundreds or thousands of contributors to a single project.[13] In our discussion of this history, we point out both the efficiency gains that provide the driving force behind the evolution of the public software institutions, as well as the incremental institutional innovations that allow for increasingly stronger coalitions.

3. The History of Public Software Institutions

In this section we turn to the history of institutions that have provided software as a public good since 1950s. We point out the role that the different sources of efficiency gains that we described above played in the development of such institutions.

13 For example, Kroah-Hartman et al (2008) report that over 3700 individual developers from over 200 companies have contributed code to the Linux kernel since 2005. This number represents only specific contributions of code to the kernel, itself just a small part of the larger Linux operating system. It does not count thousands of people involved in packaging Linux, finding errors or writing documentation.

3.1. SHARE and FORTRAN

Open source software first attracted popular attention with the rise of Linux in late 1990s and Netscape's decision to release its browser under a free software license. The practice of developing software as a public good has a long history, however, and in fact predates the earliest attempts at selling proprietary software by over a decade. We start our history of public software with a discussion of collaborative arrangements between IBM and its customers that took shape in the 1950s, before the term "software" was even invented. We use this early case to illustrate a number of incentives for contributing. We put those incentives in the context of overall efficiency. Vendors of complementary goods have a natural incentive to provide software for free. However, additional savings are possible if such software is developed collaboratively by the users, who have better knowledge of their own needs. Such users may benefit from sharing their software by receiving improvements made by other users. They can also benefit from network effects if sharing software leads to a larger network.

In 1952 IBM entered the market for digital computers, following the introduction of the first commercial computer by Remington Rand a year earlier[14] (Ceruzzi 2003). Within the first year, IBM observed that while computer hardware was extremely expensive, many customers spent as much or more programming their computers as they paid IBM for the hardware.[15] Realizing that hardware and programs were highly complementary — or, to be more precise, that programs were complementary to the specific computer models for which they were written — IBM looked for ways to expand its hardware business by lowering the cost of programming for its customers (Campbell-Kelly 2003).[16] Since many of the customers faced the same problems, IBM

---

14 An earlier computer — the ENIAC — was built in 1944 for the US Army. However, ENIAC was not offered for sale and only one ENIAC was built. The UNIVAC, built by the designers of the ENIAC by then working for Remington Rand, was built in 1951. The production of UNIVACs was increased in later years: to 2 in 1952, 3 in 1953 and more than a dozen in 1954 (Ceruzzi 2003, p. 28).

15 According to Cerruzi (2003, p. 82), a graph showing the changing ratio between software and hardware expenditures (from 1:4 in 1960s to 4:1 in 1980s) was popularized in 1970s as a marketing move by the newly emerging software companies. According to Cerruzi, later studies have found that the ratio of hardware and software expenditures have remained mostly constant — and roughly equal — over the years. (According to the US Census Bureau, US business spent roughly the same amount of money on software and hardware in 2006 — about $100 billion on each.)

16 IBM appears to have learned this lesson from its earlier experience with tabulating machines. According to

started to write some of the programs and offer them to the customers for free, thus avoiding duplication of efforts between the customers.

Early on, however, IBM also had a different idea for reducing customers' programming expenses: encouraging the clients to share software with each other. It pursued this strategy by setting up an association for its users in 1952, with a mission of helping IBM customers help each other (Campbell-Kelly 2003). The association was later renamed "SHARE," the new name explicitly representing the objective of sharing knowledge and software (Campbell-Kelly 2003, p. 33, Campbell-Kelly and Garcia-Swartz 2008). While this could be viewed as an attempt to make the customers bear the burden of programming costs, the perfect complementarity between software and hardware would have made such shifting of costs short-sighted, as it would have reduced the demand for IBM hardware. Rather, customer-led development of software offered efficiency gains. IBM users employed sophisticated teams of engineers and often had a better idea of what they needed and how to achieve it than IBM did. Putting software development in users' hands ran a risk of duplication of effort, and would have left out users with less capacity for developing their own software. The user association was created to avoid this problem, aiming to give IBM customers collectively the same economies of scale that were available to IBM.[17]

SHARE members — two dozen organizations that had purchased IBM 704 — proceeded to share existing software and to develop additional software collaboratively, building on sharing practices that predated the association.[18] The most prominent example of software produced by SHARE members was the SHARE Assembly Program (SAP) — an early version of a tool for automatic generation of a binary programs.[19] SAP was provided by United Aircraft Corporation,

---

Campbell-Kelly (2003), before entering the computer market, IBM was already used to working with customers to help them setup accounting machines applications. Rand didn't offer as much support for its Univac. As a result UNIVAC was notoriously hard to setup. There were stories of UNIVAC customers taking two years to get their UNIVAC working. Rand eventually lost its market share to IBM.

17 Note that in addition to allowing users to pool their work, SHARE enabled transfer of technology from the more experienced and demanding users — von Hippel's (1998) "lead users" — to those that had less experience. This role of SHARE is noted by Armer (1955/1980).

18 Armer (1955/1980) writes: "[Before SHARE] we 'shared' machine wiring diagrams, usually by submitting them to the machine manufacturer, who reproduced them and distributed copies to the field" (p. 123).

19 Today, "an assembly program" is understood to mean a program that converts source code written in a

and was a modification of its earlier assembly program (UASAP), which was in turn based on a system designed by General Electric and MIT.[20] Armer (1955/1980, p. 127) estimates that SAP saved SHARE members a total of about $1,500,000 in 1955 dollars.

Eyewitness accounts of SHARE history do not address the question of why United Aircraft wished to make this contribution, focusing instead on the issue that they see as a real challenge: getting share members to *accept* each other's work. In fact, SHARE members had to choose from four different assembly programs, offered by different members. According to Greenwald (2008), United Aircraft "browbeat" SHARE into accepting their assembler, threatening to leave the group if another proposal were accepted.

To understand United Aircraft's interest in getting other organizations to use their software for free, and in fact their readiness to put additional resources into modifying it to suit the needs of some of the other SHARE members, we must consider, first of all, that United Aircraft already *had* an assembly program (UASAP) that it had written for its own needs, and quite likely had made a complementary investment into software written for it. If UASAP were accepted as the standard by other members, United Aircraft could benefit from the future software that other organizations would write for it and at the same time get to keep the software that it had written for UASAP.

Strong network externalities are common in software.[21] An organization or an individual who writes (or commissions) software for their own needs may therefore benefit from letting others use it. They often stand to benefit even more if they let others modify the software and share the modifications. First, liberal licensing would make the software more valuable for potential users, leading to a larger network, and thus additional benefits to all users. Additionally, the user may benefit from improvements made by other users. For example, Stein (2007) shows

---

minimalistic programming language into binary format, and is a simpler version of what today is typically called a "compiler." The SHARE Assembly Program, however, simply assembled multiple chunks of binary codes (written by hand) into a single binary program (Salomon 1993, p. 8).

20 Shell, who himself worked for General Electric, writes: "[UASAP] was developed by United Aircraft Corporation and patterned after the first such program for the 704 which was developed under the writer's direction by the General Electric Company. This, in turn, was patterned to a considerable extent after the 'Comprehensive System' which was pioneered at M.I.T. for use on the Whirlwind computer" (1959, p. 123).

21 See Katz and Shapiro (1985, 1986) for a general discussion of network externalities.

the potential benefits of sharing even among direct competitors: individual actors may benefit from truthfully sharing underdeveloped ideas with competitors with the expectation that continued truthful exchange would increase the probability of payoff for both competitors, assuming that their markets intersect only partially.

According to Campbell-Kelly and Garcia-Swartz (2008), SHARE did not become a major institution largely because it brought together a number of competing businesses, which made the members concerned that their collaboration may be interpreted as anti-competitive collusion. This led to a decision to avoid joint development of application software, and focus exclusively on generic modules and system software — the kinds of software in which IBM's better understanding of the hardware was more important than the user's better understanding of their needs.

Aside from anti-trust concerns, many SHARE accounts note the high transaction cost of cooperation. SHARE collaboration relied on occasional meetings in different cities (Greenwald 2008), which made it hard to achieve understanding or share the improvements. IBM's centralized organization proved more efficient in providing innovation through the 1960s. The difficulty of distributing software projects was analyzed by Brooks (1975/1995), who argued that without a clear hierarchy, an increase in the number of developers working on a project leads to a quadratic increase in the cost of communication between them, which comes to dominate the linear increase in productivity. This principle, which became known as "Brook's Law," was challenged in 1970s through the introduction of new software development practices, which stressed breaking software down into small and relatively independent modules (e.g., Parnas 1972).

One of the software products developed in late 1950s by IBM itself (though with some participation of SHARE users — Backus 1979) was FORTRAN, a system that allowed generation of binary software from a format that was easier to write than the assembly language used prior to that. FORTRAN dramatically reduced the cost of software development, but at the same time transformed the economics of software development by introducing a potential to run the same program on computers of different models. In the next several years, almost all of

IBM's competitors implemented FORTRAN for their hardware (Campbell-Kelly and Garcia-Swartz 2008, p. 9), making it possible to run some of the software written for IBM computers on computers made by IBM's competitors and vice versa.[22] While much of the software continued to be written for specific computers until the 1970s, the availability of FORTRAN and similar systems made it possible for hardware manufacturers to free-ride on many of the software investments made by their competitors. As it became harder for hardware manufacturers to capture the value of their investments into the development of software, hardware manufacturers like IBM had fewer incentives to offer their software for free.

While FORTRAN promised to dramatically reduce the cost of software development, IBM did not attempt to prevent its competitors from imitating FORTRAN. Instead, IBM helped them by distributing the 1950s equivalent of source code for their FORTRAN implementation, together with detailed explanations of how it worked, making it easy for IBM's competitors to adapt it to their machines. Historians attribute such generosity to the fact that software was nor perceived as having market value (despite the recognized high cost of development!) and to the liberal "spirit of the times" (e.g., Campbell-Kelly and Garcia-Swartz 2008, p. 8). Indeed, since software was not fully covered by copyright law at the time, IBM had few options for selling FORTRAN and perhaps could not have prevented its competitors from reproducing it. However, IBM could have attempted to increase the costs of imitation. It chose to do the opposite.

The release of FORTRAN's source code made sense for several reasons. First, by releasing FORTRAN's code and documentation, IBM made it possible for some of its customers to improve FORTRAN.[23] Sharing the code with customers while keeping it away from competitors was probably not viable due to high transaction costs. Second, while FORTRAN was the first practical product of its kind, IBM's competitors were working on similar products. One

---

22 Note, though, that COBOL, released in 1960 and based on an earlier language developed by Remington Rand, was the first language to make it possible to execute the same program on two different computers (Campbell-Kelly 2003, p. 34-36, Ceruzzi, p. 92). Around the same time people start working on other ways to run IBM software on non-IBM machines. In 1963 Honeywell introduces "the Liberator" - a program that allowed users to convert software written for IBM 1401 to Honeywell 200 (Campbell-Kelly 2003, p. 98). But it was the standard languages like FORTRAN and COBOL that really helped commodify the hardware.

23 See for example, Davidson (1984) for a discussion of modifications introduced at the University of Wisconsin in 1961.

of them — COBOL — had the support of the US government, which promised to not purchase or lease equipment that did not support COBOL (Ceruzzi 2003, p. 92). This mandate can be understood as an attempt to by the government to avoid the hold-up problem that it would face by investing resources into development of software that could only be used with specific manufacturer's hardware. (The government also had a legitimate reason to worry about the underinvestment by other private actors due to the risk of hold-up.) IBM therefore had an interest in making FORTRAN software work with non-IBM hardware to avoid the risk of making COBOL a more attractive option for its customers. This was amplified by the fact that FORTRAN and COBOL both promised to create strong network effects, that could create disproportionate benefits for the originators of the technology.

Starting in the 1960s, hardware manufacturers faced fewer incentives to provide software for free, though this effect was somewhat reduced by IBM's increasing monopolization of the hardware market. This created opportunities for specialized firms that provided programming services to computer users. As software was increasingly recognized as a potential object of market transactions, software contractors realized that they could offer standard software for sale, starting with Autoflow — a flowchart program offered for sale in 1964 (Campbell-Kelly 2003). Since copyright law and patent law did not provide effective protection for software at the time, such vendors typically relied on trade secrets and non-disclosure agreements to make their software excludable (Menell 2002).

In 1969, antitrust action brought against IBM by the Justice Department led the company to unbundle its software operation from its hardware business, and to start charging customers for the software. This move gave a major impetus to the proprietary software industry. Even as IBM started to charge customers for software, it continued to provide them with source code and allowed them to modify and share it until the early 1980s (Campbell-Kelly and Garcia-Swartz 2008).

3.2. ARPANET and Unix

In this section we look at a different set of institutions for providing public software, which started to emerge in the 1960s and the 1970s. Those institutions were quite different from those that IBM tried to create in the 1950s. Production of public software came to rely heavily on the support of the US government, which funded academic and military researchers. An important contribution was also made by AT&T, a regulated monopoly. Provision of public software started to use a system of incentives borrowed from academic institutions. The projects described in this section also provided technological means for reducing the cost of collaboration, making it easier to create public software.

Industry-led collaborative efforts such as IBM SHARE produced a body of public software in the 1950s and 1960s, but did not result in lasting institutions for provision of public software, in part due to anti-trust concerns. From the mid-1960s, software was increasingly provided as a private good, despite the weak level of protection offered to it by law at the time. By 1970s, however, alternative institutions started to emerge as providers of public software, supported to a large extent by the US Government and AT&T — a highly regulated public utility. The most notable of the government sponsored projects was ARPANET, which attempted to build, starting in 1967, a decentralized network for computer-to-computer communication. This project was later extended to what became known as the Internet. The work was funded by ARPA,[24] an agency of the Department of Defense, but was executed through collaboration of several teams of university researchers and computer companies. After some initial uncertainty about the legal status of the software created by the project, ARPA made a decision to require its contractors to make the source code of all ARPANET software freely available.[25] This move

24 ARPA was renamed to "DARPA" in 1972. We use the name appropriate for each historical period.

25 Abbate (2000) writes: "To preserve its strategic advantage in having designed IMP, BBN [one of the firms contributing to ARPANET] tended to treat the IMP's technical details as trade secrets. In addition, Heart was worried that, if BBN shared too much information about the IMP, graduate students at the host sites would try to make unauthorized 'improvements' to IMP software and would break havoc on the system. One of the more heated conflicts within the ARPANET community arose when BBN refused to share the source code for the IMP programs with the other contractors, who protested that they needed to know how the IMPs were programmed in order to do their own work effectively. The authorities at ARPA eventually intervened and established that BBN had no legal right to withhold the source code and had to make it freely available." (p. 76-77.)

facilitated improvement of the software and protected ARPA from hold-up.[26]

ARPA's second decision was to rely on academic researchers in addition to the industry contractors. By involving academic researchers, ARPA brought into software development some of the institutions that underly the provision of science. Production of science is similar to production of software in that its utility can be maximized and its total cost can be minimized if scientists share their intellectual products without attempting to make them excludable. Over time, a set of institutions developed that made it possible to provide science as a public good. Such institutions enabled the scientific community to obtain funding from both public and private sources, relying on a complicated mix of altruistic and self-interested motives by the sponsors. They also provided methods for evaluating scientific output, the most important of those being peer review (Scotchmer 2006), which partially compensates for the lack of a market selection. While modern open source institutions diverge substantially from scientific institutions, both in how they attract external funding and in their handling of credit, they share important characteristics and have clear historical links, the ARPANET project being an important opportunity for a transfer of institutions.[27]

Historians of computing explain ARPA's decision to rely on "collegial" working arrangement by the fact that people who made those decisions have themselves come from academia, and the fact that ARPA needed assistance of academic researchers and found that collegial arrangements helped it obtain their cooperation (Abbate 2000). To put this differently, however, the ARPA administrators faced a need to provide a public good and relied on a known solution for it, which also allowed the scientists to collaborate with ARPA without detriment to their academic careers.

Like SHARE members, ARPANET contributors were distributed among a number of sites, located in such places as Boston, Pittsburgh, the San Francisco Bay Area, and Los Angeles. They thus potentially faced the same high costs of collaboration. ARPANET managed to reduce

26 Since ARPA was a government agency, its decision to make the software public could be alternatively understood simply as a matter of public provision of a public good.

27 In open source as in science the contributors may be willing to accept lower wages in exchange for having the output of the work be publicly available. The motives may include career concerns as well as ego gratification. Stern (2004) shows that young scientists accept lower wages in order have increased opportunities to publish.

those costs, however, using the very technology that it was building. The computer networking technology produced by ARPANET "itself provided a new way to coordinate dispersed activities and came to function as a meeting place for the computer science community" (Abbate 2000, p. 54). This involved use of such tools as email, developed by ARPANET researchers in 1972 (p. 108).

ARPANET was one of the most notable, but not the only project that involved "collegial" production of public software in 1960s and 1970s and was funded by the US government. ARPA funded numerous projects in many universities, often without asking for very specific deliverables, but rather looking for overall innovation (Abbate 2000, Weber 2005). While the most important product of the ARPA-funded projects were ideas expressed in academic publications, they also produced a large quantity of software, much of which was distributed without restrictions. One small but important product was Emacs, a text editor designed by an MIT programmer named Richard Stallman. Stallman distributed Emacs for free and allowed others to make changes, but demanded that any improvements made to Emacs would be returned to him so that he could incorporate them into Emacs and distribute them to all users. This policy, which Stallman called "communal sharing," represented an important institutional innovation. While it proved problematic in its original form, it served as a stepping stone towards what later became known as "copyleft" licensing.

Among the less successful projects funded by ARPA in 1960s was a new operating system called "Multics." Multics promised to deliver features that were of interest to AT&T and that IBM had been willing to provide in its software (Ceruzzi 2003, p. 156). AT&T's Bell Laboratories then joined the Multics project with the hope of accelerating it and steering it closer to AT&T's needs (Ceruzzi 2003, p. 155, also Salus 1994, p. 26). The Multics project failed, likely due to its excessive complexity. However, a number of Bell Labs researchers produced a simplified version of Multics which they called "Unix." Unix proved an immediate success.

In 1960s and 1970s, AT&T was operated as a regulated monopoly. Because of an earlier anti-trust decree, AT&T was banned from engaging in any business other than telephone service. The company's lawyers understood this to include a prohibition on selling software. While

AT&T was allowed to engage in research and obtain patents, it had to license all patents for nominal fees. Unable to sell Unix, AT&T decided to license it for a nominal fee, as it had done with patents and other software before (Salus, p. 56-60). While Unix research brought no direct income to the company, AT&T was a regulated monopoly and could pass its research costs to the consumers, as long as such costs could be justified to the regulators.

In 1973 Unix was re-written in C, a new language also developed at AT&T. While FORTRAN and COBOL had already enabled portability for certain kinds of software, a combination of C and Unix made it possible to write almost any software in a way that would allow it to be moved between computers. Doing so, however, required access to the source code, because different binary software had to be generated for each machine and modifications to the source code were often required. Being able to move software between computers also meant being able to use it on *later* computers, increasing the time horizons for software users. While much of the software written in 1960s (including all of the ARPANET software) was hardware-specific and could be expected to become obsolete eventually, most of the software written for Unix in 1970s can be used today.[28] Increased time horizons made access to source code even more important, as some users could now plan to use their software for decades.

Unix introduced an important technical innovation that helped defeat Brook's Law and paved the way for larger scale collaboration. Learning from the failure of Multics, Unix designers took a minimalistic approach, designing the system as a collection of simple tools, each designed to serve a specific purpose, and providing a mechanism for chaining such tools together for more complex task. This approach reduced the amount of interdependencies in the system, making it possible for a larger number of people to collaborate effectively.[29]

Researchers at several universities that licensed Unix from AT&T made improvements to Unix, both improving tools provided by AT&T and writing new ones. The Computer Systems Research Group (CSRG) at the University of California, Berkeley emerged as a particularly

28 This software would in fact be still in active use today, if not for the legal reasons that we discuss below, which led to a re-implementation of much of it in the 1980s.

29 The notion of "decomposition" was quite new at the time. Parnas (1972) is considered the seminal paper on this topic. The first Unix paper was published in 1974 (Ritchie and Thompson 1974).

important site of such work, performed as a matter of research activities. In 1977, a Berkeley graduate student started compiling tapes of Berkeley modifications to Unix and offering them at nominal cost to other researchers. This collection of software, soon expanded to a full operating system based on AT&T's Unix but modified to run on new hardware, became known as "Berkeley Software Distribution" or "BSD." CSRG charged a small fee for the BSD tapes ($50 in 1978, according to Salus 1994, p. 143), but allowed the recipients to make copies and share them with others.

In late 1970s, Unix attracted attention of DARPA (formerly "ARPA"), which was looking for a unified base for its projects, including the emerging Internet. DARPA was in particular attracted to the fact that Unix code was available (Weber 2005, p. 34).[30] DARPA offered support to Berkeley CSRG to extend BSD for DARPA's needs. Funded by DARPA, Berkeley researchers made many further improvements to BSD and integrated into it networking code originally developed by a DARPA contractor. The new version of BSD, with improvements commissioned by DARPA, was released in 1983 and proved to be an immediate success, turning BSD Unix into a foundation of the Internet (ibid, p. 35).

The future of BSD, however, was soon clouded by the uncertainty of the property rights around it. While the use of BSD was always assumed to require a license for the original Unix, this restriction was not important, as AT&T had been offering the license at nominal prices. Around 1980, however, several changes led AT&T to reconsider this policy and to try to treat Unix as a proprietary software product. This move created serious challenges to public software, since AT&T was not only making a claim to a specific software product, but was asserting control over what was emerging as a preferred platform for public software development. As we will see, the public software institutions managed to overcome this challenge through a number of institutional innovations.

30 While avoiding hold-up would be an obvious reason for DARPA to choose software with open code, Weber mentions a rather different reason: the researcher's lack of interest in working with closed-source solutions (p. 34).

3.3. The New Copyright Regime

Around 1980, the legal protection for software in the United States underwent an important change. Prior to 1964 software was understood to be outside the bounds of copyright law and was offered no protection beyond trade secrecy. In 1964 the Copyright Office extended copyright protection to software, giving software producers a new mechanism for making software excludable. This mechanism suffered from an important drawback, however. As other works, software had to be *published* and *registered* to be protected by copyright. Specifically, *the source code* of the software had to be published and deposited with the Copyright Office (Menell 2002). Software producers could thus protect their work using either copyright or trade secrets, but not both. As we discussed earlier, legal restrictions cannot make software effectively excludable if the source code is shared. To make software truly excludable, the producer needs dual protection: trade secrecy for the source code (which is never revealed), and copyright protection for the binary that is offered on the market. A 1980 amendment to the Copyright Act of 1976 offered software producers exactly this protection.[31]

This change created obvious incentives to trade in binary software. The growing software industry in turn attracted some of the people who were earlier working on public software. Bill Joy, one of the key people behind BSD, left Berkeley in 1982 to co-found SUN Microsystems, which used BSD as the foundation for its own proprietary Unix variety (Salus 1994). Many members of MIT's AI Lab similarly left to start private ventures in early 1980s (Levy 1984/2001).

The legal changes, however, also transformed the practices of those who continued to provide public software by making the public status of such software more explicit. Prior to 1980, if the code had not been registered with the Copyright Office (which was rarely done), its very

---

31 The 1976 act, which took effect in 1978, extended copyright protection to unpublished works, but did not cover software explicitly. The issue of software protection was then investigated by a congressional committee, which issued a report recommending copyright as the preferred form of protection for software (CONTU 1978). This suggestion was implemented with an amendment in 1980 (Menell 2002, p 18). The same year US Congress passed Bayh-Dole Act, which allowed universities to get patents for inventions created with the use of federal funds. This did not have any immediate effect on public software, however, patents were rarely used to protect software until 1990s (Graham and Mowery 2003).

transfer removed the only other form of protection — trade secrecy. Simple sharing of the source code thus resolved the hold-up risks. From 1980, however, a party could offer source code to others yet keep all rights to it. To avoid hold-up, the recipient had to obtain an explicit license for the code. Responding to this change, programmers started sharing code with explicit licenses, which consisted first of simple notes stating that the code was "in public domain," but gradually evolved into more formal licenses, with proper legal terms and safeguards such as the disclaimers of warranty.

The move towards keeping public software safe from proprietary claims was also encouraged by a change in AT&T's position on Unix. In 1978 the US government brought another antitrust case against the company, leading to a settlement in 1982, in which AT&T agreed to divest of its local telephone exchange business. As a part of the settlement, however, AT&T received a permission to sell software (Salus 1994, p. 190). The company attempted to use this right to sell Unix, offering a version of Unix to clients for increasingly high license fees.[32] Such fees have often been described as "prohibitive" (e.g., Weber 2005), a term that may suggest deadweight loss. It may be more appropriate to say, however, that AT&T's increasing licensing fees demonstrated its position as a profit-taker, holding up those who had contributed improvements to Unix or made complementary investments. While the eventual court case between AT&T and the University of California (*USL v. BSDi*, 1992–1993) was settled in favor of the University, this outcome was uncertain at the time.

The new BSD promised great value to the users, offering them a system capable of connecting to the Internet, with full source code. Such benefits, however, could be fully captured by AT&T, which asserted that use of BSD required an AT&T license. Since most of BSD code had been contributed by Berkeley, replacing AT&T's code with new code would be substantially less expensive than writing an equivalent of BSD from scratch. It was still a rather monumental task — more than CSRG could take on — but it could be done if the potential beneficiaries could form a coasian coalition and pull their efforts. In 1989, Keith Bostic, one of CSRG staff members, decided to try to build such a coalition, asking BSD users for help in replacing AT&T

32 $100,000 in 1988 according to Weber (p. 39).

code.

The large number of beneficiaries (CSRG's 1988 release of parts of BSD led to over a thousand requests for copies at $1000 processing fee), however, created an obvious free-rider problem. Bostic attempted to solve this problem by setting up matching contributions: convincing some of the CSRG staff to agree to contribute the most difficult part of the system (the kernel), if Bostic could find contributors for other parts.[33] Bostic then recruited contributors over the Internet and in user meetings, telling them that an effort to replace AT&T code was underway but that help was needed. He offered no rewards other than inclusion in the list of contributors — and an opportunity to help create a public good of obvious value. While Bostic's call invited people to contribute their time for largely altruistic reasons, potential contributors knew that the effect of their work would be amplified by the fact that it was unlocking the value of the software that had already been written by CSRG.[34] The strategy proved successful: within eighteen months, Bostic collected most of the necessary utilities, after which CSRG staff re-wrote the kernel (McKusick 1999). The complete system was released in 1991. AT&T Bell Labs sued Berkeley to stop its release but lost. The case closed in 1993 with Berkeley's right to release BSD upheld.

The crisis of public software in the 1980s also led to a different and more radical response, which introduced important additional innovations into the public software institutions. In 1983, Richard Stallman, the programmer at MIT who had earlier led development of Emacs, announced his plan to write a completely free version of Unix, which he called "GNU" (a recursive acronym for "GNU's Not Unix"), launching what became known as "the Free Software Movement." Stallman's announcement of his plan expressed his motivation in *ethical* rather than economic terms: "I consider that the golden rule requires that if I like a program I must share it with other people who like it. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. So that I can continue to use computers without violating my principles, I have decided to put together a sufficient body of free software so that I will be able

33 See Guttman (1978) for a discussion of matching contributions.
34 Effect of matching contributions on public good provision was studied Guttman (1978), Andreoni (2006), and Ghosh and Mahdian (2008).

to get along without any software that is not free." (Stallman 1983). In other words, Stallman believed that it was unethical to refuse to share with others tools he used himself. Since copyright law prohibited sharing proprietary software, Stallman believed that he had to avoid using it, and instead dedicate his time to providing an alternative.

Stallman's manifesto provided a radical articulation of some of the assumptions of the public software culture. A well articulated ideology has proven to be useful to public software institutions, as it helped the members organize for collective action. While earlier the production of public software drew primarily on the institutions of public science, Stallman's approach drew on an entirely different set of institutions: political movements. Unlike scientific institutions, which seek public funding for provision of a public good, political movements aim to provide a type of public good (positive social change) by seeking contributions from individuals who believe in the cause advocated by the movement. While many contributors to public software have rejected Stallman's "political" approach, for many others political motivations are an important reason for contributing. According to Ghosh (2005), a survey found that 30% percent of contributors to open source projects reported a belief that "software should not be a proprietary good" as one of the reasons for contributing to public software.

An important element of Stallman's ideology was a stress on a libertarian rationale over the simpler issue of cost. Stallman repeatedly told his supporters that "free software is a matter of liberty not price," and asked them to think of "free speech" rather than "free beer." He stressed the importance of giving the users freedom to adapt software for their needs, improve it and share the improvements, explaining that "being free to do these things means (among other things) that you do not have to ask or pay for permission" (Stallman 1996/2002, p. 41). Stallman links this notion of freedom to aggregate efficiency, by pointing out the transaction costs inherent in having to ask for permission and the resulting duplication of effort (Stallman 1984/2002). He also points out the danger of hold-up, stressing that with free software "users will no longer be at the mercy of one programmer or company which owns the sources and is in sole position to make changes" (Stallman 1984/2002, p. 34), though he focuses only on the individual cost of hold-up and does not make a link between holdup and inefficiency due to underinvestment.

Stallman's rethinking of public software as a political movement led him to seek independence from existing institutions. In a major diversion from earlier public software practices, Stallman resigned from MIT and ran his project independently, relying on private consulting and Emacs sales to support himself.[35] (MIT supported him indirectly by offering a computer and an office.[36]) Stallman then set up a non-profit foundation (Free Software Foundation) to support the project. Non-profit foundations have since become a key element of public software institutions, making it possible for public software contributors to attract financial resources from a wide range of actors, thus achieving independence vis-á-vis other institutions (O'Mahony 2005). By 1988, Free Software Foundation started to attract substantial cash and hardware donations from a range of actors, who also loaned Stallman their programmers. The early donors included Sony and Hewlett Packard — both of them hardware manufacturers that stood to gain from the Foundation's efforts to create a free complementary good.[37]

Stallman's third innovation aimed to protect public software from companies that might make modifications and then treat the improved software as private. He attempted to do with a peculiar licensing solution, which became known as "copyleft" licensing, based on the "communal sharing" arrangement he earlier attempted with Emacs. Instead of declaring GNU software to be "in public domain" or using what later became the BSD license (which, in its current version, essentially allows unrestricted use, modification and redistribution), Stallman and his followers offered all recipients of their software the right to modify and redistribute it, but imposed an important restriction. If the user chose to redistribute GNU software, whether in its original or modified form, they had to disclose the source code and to share it under the same

35 Stallman offered Emacs for a price of $150, but allowed users to make unlimited copies and modifications. The scheme brought him up to $1000 a month (Moody 2001). Such submissions could be understood as donations masqueraded as purchases: Stallman's supporters could essentially make their employer donate to Stallman by purchasing Emacs instead of seeking a free copy.

36 This offer was particularly important as Stallman was allowed to sleep in this office, as he proceeded to do for twelve years, according to Moody (2001).

37 The GNU project reported its first large donation — $10,000 from Sony — in February 1988. Sony also lent GNU a programmer for six months, and provided two Sony workstations (GNU 1988, 1989a, 1989b). The project then reported two donation of $100,000 — one from Hewlett Packard and another from "Anonymous Contributor" — in 1989 (GNU's Bulletin 1989, #7). Additionally, Stallman received the MacArthur Genius Award for his work in 1990 which provided him with $250,000 (Williams 2002).

license as the original software. Any software based on copyleft software would thus have to retain its copyleft status.[38]

The GNU Project produced a lot of software that is used to this day but was never completed in the form in which Stallman envisioned it, having ran into a serious technical problem in early 1990s. The project had everything but the "kernel" — the centerpiece of an operating system. Stallman wanted to use a particular type of kernel called "microkernel," which was widely considered to be superior but turned out to be difficult in practice. In 1992, however, Linus Torvalds, an undergraduate student living in Finland, decided to write an alternative kernel for GNU, basing his kernel on another one, called "Minix." His kernel, which became known as "Linux," was a "monolithic" kernel — an approach that was derided by many people, including the author of Minix. Linux worked quite well in practice, however, and Linus managed to attract many volunteers who helped improve it. The combination of GNU and Linux produced a complete operating system. This combination came to be known as "Linux," though purists call it "GNU/Linux."

While Torvalds's efforts were originally disregarded by Stallman, GNU's free software license gave Torvalds the right to pursue his project without Stallman's approval. By granting others unconditional right to modify the software, Stallman made it possible for others to correct his mistakes (over his disapproval) and to complete the project.

3.4. Public Software Goes Corporate (Again)

In addition to completing GNU, Linux introduced an important additional change. As nearly all Unix-like systems of the time, Stallman's GNU was intended to run on "workstation" computers, which were considered cheap in comparison to many other computers of the time, but were an order of magnitude more expensive than the emerging personal computers. ($30,000

38 Note that copyleft license does not require that the users share their modifications. It gives the licensee a choice of either not sharing their modifications with anyone or releasing them under the same terms as the original authors. What they cannot do is distribute the software to others with additional restrictions. See Lerner and Tirole (2005) for a discussion of factors that may influence developer's choice of a public software license with and without a "copyleft" clause.

dollars is a typical price quoted for a workstation around 1990. By contrast, personal computers typically cost under $3,000 at the time.) The price of workstations was hardly prohibitive for programmers like Stallman, Bostic and others, who had access to university machines (which were often purchased with public funds or provided by wealthy donors). Linus Torvalds, an undergraduate student at the time, was only in possession of a $3,000 personal computer. His goal was to make a Unix-like system run on it.

Personal computers had been introduced in late 1970s, but presented little competition for the larger machines until the early 1990s. Such computers were small and could not perform functions of interest to DARPA- and AT&T-funded Unix programmers. In 1980s, following the introduction of the IBM PC, personal computers spread in popularity, but did so among non-technical users, who typically used them for word processing and games. Access to the source code offered little benefit to such users, who instead put the highest premium on ease of installation and use. Additionally, unlike the diverse world of workstations, the PCs quickly converged to just two processor types, which made it possible to run the same binary software on most PCs. Consequently, the PC software industry quickly converged to a proprietary model. An important exception to this rule was shareware — software distributed with a license that allowed unlimited copying, but accompanied with a request for donations. Shareware benefited from certain gains in efficiency — the distributors saved on marketing and could achieve large network effects by including users that were not willing to pay for the software. However, shareware producers typically did not share the source code and thus did not benefit from all the efficiencies associated with public software.[39]

By adapting GNU's software for his personal needs, Torvalds introduced GNU to a pool of users that had been overlooked until then: technical users of personal computers. Linux was not the first Unix-like system for personal computers, but it differed in an important respect from the two that preceded it: Minix, developed as an instructional tool by a university professor, and

39 The concept of shareware was introduced in early 1980s. Curiously, one of its inventors was a former Microsoft programmer Bob Wallace. Wallace later said that he thought that letting people use the software would help them understand why it is useful and that would help the software reach more users. Wallace made some money with shareware, building a business that brought in around $2,000,000 in revenue per year at one point (Callahan 1990).

386BSD, a PC version of BSD. Minix and 386BSD were both popular among PC users, but suffered from many deficiencies. Individual users attempted to contribute both ideas and actual modifications, often fixing specific problems that they encountered. (Both systems were distributed with source code and allowed modifications.) Such modifications, however, were often rejected by the original authors who had different plans for the software.[40] Torvalds, on the other hand, tried to capitalize on the users' interest in submitting improvements. The credits for Linux 1.0, released in 1994, include eighty names. By contrast, the BSD kernel was written by a handful of people at Berkeley.

The availability of Linux offered many companies an opportunity to use cheap personal computers for functions that earlier required more expensive equipment. A number of firms arose to facilitate this, offering business clients a more polished version of the GNU/Linux system. This change resembled the proliferation of Unix companies such as Sun Microsystems in early 1980s, but differed in a crucial way. Unlike Sun Microsystems, which had an opportunity to distribute an improved version of BSD as proprietary software, the Linux companies of the 1990s were constrained by the "copyleft" license terms of GNU/Linux, which required them to provide their customers with the source code and to allow further modification and distribution. While this limited the expected profits of such ventures and reduced the incentives for entry, private underinvestment was partly made up by the development of the non-profit model and appearance of foundations similar to Stallman's (see O'Mahony 2005).

While Stallman's licensing terms proved crucial for the growth of public software in the 1990s, the ethics-based ideology of the "free software" movement was limiting the appeal of public software for many people involved in the software industry and was impeding private investment. In 1997, Eric Raymond, a Unix programmer, wrote an essay arguing for sharing the source code by stressing the efficiency gains rather than ethics.[41] The essay was read by the management of Netscape Corporation, which was at the time being driven out of its web browser

40 In case of Minix, Tanenbaum wanted to keep the system simple enough to work as an instructional tool (Tanenbaum 2004). (Linux was largely based on Minix design.) In case of 386BSD, the reluctance to accept users' modifications stemmed more from author's desire to build the system the right way (Raymond 1999/2001).

41 The essay was later reprinted as a chapter ("Cathedral and the Bazaar") in Raymond (1999/2001).

market by Microsoft's Internet Explorer, offered for free (but without the source code) to all users of Windows. The essay convinced Netscape to release the source code of its browser in order to enable user-driven innovation and to compete with the Internet Explorer on quality rather than price, while making money on complementary goods. Following Netscape's decision, Raymond convened with other open source enthusiasts to discuss the opportunities offered by Netscape's decision. The meeting concluded with a realization that public software would benefit from a "marketing campaign" and "that it would require marketing techniques (spin, image-building, and rebranding) to make it work" (Raymond 1999/2001). As a re-branding effort, the meeting participants decided to start using the term "open source" instead of "free software," believing that the new term would make it easier for corporate America to embrace public software. Having built a coalition based on altruistic contributions by participants motivated by shared ideology, public software enthusiasts were now attempting to broaden it by including a new type of actor: software companies.

Netscape's decision to "open source" their browser did succeed in giving the product a second life, attracting contributions from various actors that were either interested in having a browser for specific platforms or in maintaining competition in the browser market. The open source browser based on Netscape's code, first released as "Mozilla" and later as "Firefox," currently serves about 20% of the browser market and was a major factor in reducing Internet Explorer's share from 95% in 2002 to 71% in 2009. Since 2003 the development of Firefox has been coordinated by a non-profit Mozilla Foundation, which currently receives most of its funding from "a search engine provider," typically understood to be Google.[42]

Since 1998, development of "open source" software has been increasingly supported by private companies, which discovered that open source often offers a cost-effective way to create or improve products complementary to their core business. In 2001 IBM announced its plans to contribute $1 billion to Linux and later reported that the investment paid off in a single year. In 2007, Sun released as open source the implementation of Java — by some counts the most

42 According to the Foundation's 2006 report, the foundation received 85% of its 2006 revenue from "a contract with a search engine provider for royalties" (p. 11). Those are widely assumed to be fees paid by Google for making Google search the default option in Mozilla Firefox.

popular programming language in use today. In 2008, Nokia did the same with Symbian, a popular operating system for mobile phones.

In some ways, public software has made a full circle, once again coming to rely heavily on contributions of large software companies. At the same time, the institutions that have supported public software have undergone a substantial evolution. In the 1950s, collaborative projects such as SHARE arose spontaneously in situations where the efficiency gains from sharing software code were clear, and especially where small groups of potential beneficiaries could be brought together to negotiate the sharing of costs. Such coalitions often fell apart when the circumstances changed. Today, open source projects often attract massive numbers of participants, who contribute their resources for a wide variety of reasons, which include both altruistic and self-interested motivations. This is made possible by a highly evolved system of institutions. Such institutions involve licensing arrangements that prevent many forms of opportunistic behavior,[43] new organization forms such as non-profit open source foundations, and new technical means for reducing the cost of collaboration.

4. Conclusion

This paper has presented a historical evolution of the provision of software as a public good. As we have shown, provision of public software has relied and continues to rely on a wide range of individual incentives. Some actors contribute "just for fun," some do it as an altruistic act (sometimes driven by a larger political vision), some seek to sell complementary products, some simply need specific software functionality others are unwilling to provide. Moreover, different reasons have historically been prominent. Looking at the provision of public software in each decade since the 1950s, would lead one to stress different individual incentives.

Instead of focusing on the individual incentives *per se,* it is important to consider the

43 The GPL has gone through a number of revision, each attempting to either enable its wider use or to prevent specific ways of circumventing the copyleft provisions. Additionally, a number of other licenses have appeared that give certain parties specific other rights. For example, the license used by Mozilla was similar to the GPL but exempts Netscape from the copyleft requirements, making it possible for Netscape (and now AOL) to distribute a proprietary version of the browser.

larger driving force responsible for the provision of public software: the substantial aggregate efficiency gains afforded by distribution of source code rather than just the binary software, combined with the inherent difficulty of trading source code as an excludable good. As was pointed out in the literature, those efficiency gains arise in part because open source model enables a more efficient production of software. We point out an additional source of efficiency gains associated with open source. Use of proprietary software introduces a risk of hold up, making the buyer reluctant to make all the complimentary investments that may be necessary to capture the full benefit provided by the software. This can lead to underinvestment, especially when the use of the software requires large complementary investments, as, for example, is the case with computer programming language implementations, a niche in which public software reigns today.[44] Looking at the half century history of public software provision shows us how those two factors have motivated producers and users of software to look for institutional arrangements that would enable provision of software as a public good.

5. Bibliography

Abbate, J. (2000) *Inventing the Internet,* Cambridge, MA: The MIT Press.

Andreoni, J. (2006) Leadership giving in charitable fundraising, *Journal of Public Economic Theory,* 8(1), 1–22.

Athey, S. and G. Ellison (2006) The dynamics of open source movements, a working paper.

Armer, P. (1955/1980) SHARE — A eulogy to cooperative effort, *Annals of the History of Computing,* 2(2), 122–9.

Backus, J. (1979) The history of FORTRAN I, II, and III, *Annals of the History of Computing,* 1(1), 21–37.

Bagnoli, M. and B.L. Lipman (1989) Provision of public goods: Fully implementing the core through private contributions, *Review of Economic Studies,* 56, 583–601.

Bergstrom, T., L. Blume, and H.R. Varian (1986) On the private provision of public goods, Department of Economics, UCSB. Ted Bergstrom. Paper 1986B.

Bliss, C. and B. Nalebuff (1984) Dragon-slaying and ballroom dancing: the private supply of a public good, *Journal of Public Economics,* 25, 1–12.

Brooks, F.P. (1975/1995) *The Mythical Man-Month: Essays on Software Engineering*, 2nd

44 Of the top ten programming languages (as ranked by TIOBE 2009), all have an open source implementation, and for most the open source implementation is the one used most frequently.

edition, Addison-Wesley Professional.

Callahan, M.E. (1990) *Dr. File Finder's Guide to Shareware,* Mcgraw-Hill.

Campbell-Kelly, M and D.D. Garcia-Swartz (2008) Pragmatism not ideology: IBM's love affair with open source software , SSRN Working Papers, http://ssrn.com/abstract=1081613, last retrieved on October 30, 2008.

Campbell-Kelly, M. (2003) *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry,* Boston, MA: The MIT Press.

Ceruzzi, P. (2003) *A History of Modern Computing,* 2nd edition, Cambridge, MA: The MIT Press.

CONTU (1978) *The Final Report of the National Commission on New Technological Uses of Copyrighted Works*, Washington: Library of Congress.

Corbet, J. (2007) Who wrote 2.6.20? *LWN.net*, http://lwn.net/Articles/222773/, last accessed October 30, 2008.

Davidson, C. (1984) The emergence of load-and-go systems for FORTRAN, *Annals of the History of Computing,* 6(1), 35–37.

Ghosh, A. and M. Mahdian (2008) Charity auctions on social networks, working paper, Yahoo! Research.

Ghosh, R.A. (2005). Understanding free software developers: findings from the FLOSS study, in *Perspectives on Free and Open Source Software* by J. Feller, B. Fitzgerald, S.A Hissam and K.R. Lakhani, eds, Cambridge, MA: The MIT Press, 23–46.

GNU (1988) GNU's Bulletin, 1(4), http://www.gnu.org/bulletins/bull4.html, last accessed October 30, 2008.

GNU (1989a) GNU's Bulletin, 1(5), http://www.gnu.org/bulletins/bull5.html, last accessed October 30, 2008.

GNU (1989b) GNU's Bulletin, 1(7), http://www.gnu.org/bulletins/bull7.html, last accessed October 30, 2008.

Graham, S.J.H and D.C. Mowery (2003) Intellectual property protection in the US software industry, in *Patents in the Knowledge-Based Economy,* by W.M. Cohen et al. Washington, D.C.: National Academies Press.

Greenwald, I. (2008) RAND Recollections, http://users.adelphia.net/~irwingreenwald/memoir.html, last retrieved August 5, 2008.

Guttman, J.M. (1978) Understanding collective action: matching behavior, *American Economic Review,* 68, 251–255.

Hars, A. and S. Ou (2001) Working for free? – Motivations of participating in open source projects, *Proceedings of the 34th Hawaii International Conference on System Sciences*.

Hart, O. and J. Moore (1990) Property rights and the nature of the firm, *Journal of Political Economy,* University of Chicago Press, 98(6), 1119–58.

IBM (2008) Fact sheet & background: Roadrunner smashes the petaflop barrier, IBM press release, http://www-03.ibm.com/press/us/en/pressrelease/24405.wss, last accessed

October 31, 2008.

Johnson, J.P. (2002) Open source software: private provision of a public good, *Journal of Economics and Management Strategy,* 11(4), 637–62.

Johnson, J.P. (2006) Collaboration, peer review and open source software, *Information Economics and Policy,* 18, pp. 477-497.

Katz, M. and C. Shapiro (1985) Network externalities, competition and compatibility, *American Economic Review*, 75 (3), pp. 424-440.

Katz, M. and C. Shapiro (1986), Technology adoption in the presence of network externalities, *Journal of Political Economy*, 94, pp. 822-841.

Klein, B., R.G. Crawford and A.A. Alchian (1978) Vertical integration, appropriable rents, and the competitive contracting process, *Journal of Law and Economics,* 21(2), 297–326.

Kroah-Hartman, G., J. Corbet, and A. McPherson (2008). "Linux Kernel Development (April 2008): How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It" http://www.linuxfoundation.org/publications/linuxkerneldevelopment.php. Last retrieved October 30, 2008.

Lai, E. (2008) Size matters: Yahoo claims 2-petabyte database is world's biggest, busiest, Computerworld, http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9087918, accessed on November 1, 2008.

Lakhani, K.R. And R.G. Wolf (2005). Why hackers do what they do: understanding motivation and effort in free/open source software projects, in *Perspectives on Free and Open Source Software* by J. Feller, B. Fitzgerald, S.A Hissam and K.R. Lakhani, eds, Cambridge, MA: The MIT Press, 3–22.

Lerner, J, P. Pathak and J. Tirole (2006) The dynamics of open-source contributors, *American Economic Review,* 96(2), 114–118.

Lerner, J. and J. Tirole (2002) Some simple economics of open source, The Journal of Industrial Economics, L(2), 197–233.

Lerner, J. and J. Tirole (2005) The scope of open source licensing, *Journal of Law Economics and Organization,* 21(1), 20–56.

Levy, S. (1984/2001) *Hackers: Heroes of the Computer Revolution,* Penguin Books.

McKusick, M.K. (1999). Twenty years of Berkeley Unix: from AT&T-owned to freely redistributable, *Open Sources: Voices from the Open Source Revolution,* by By Chris DiBona and Sam Ockman, eds, O'Reilly.

Menell, P.S. (2002) Envisioning copyright law's digital future, SSRN Working Papers, http://ssrn.com/abstract=328561, last accessed on October 30, 2008.

Moody, G. (2001) *Rebel Code: The Inside Story of Linux and the Open Source Revolution,* Cambridge, MA: Perseus Publishing.

Mozilla Foundation (2006) Independent auditors' report and consolidated financial statements, http://www.mozilla.org/foundation/documents/mf-2006-audited-financial-statement.pdf,

last accessed April 21, 2009.

North, D. (1981) *Structure and Change in Economic History,* New York: Norton and Company.

North, D. (1990) *Institutions, Institutional Change and Economic Performance*, Cambridge: Cambridge University Press.

O'Mahony, S. (2005). Non-profit foundations and their role in community-firm software collaboration, in *Perspectives on Free and Open Source Software* by J. Feller, B. Fitzgerald, S.A Hissam and K.R. Lakhani, eds, Cambridge, MA: The MIT Press, 23–46.

Ornstein S. (2002) *Computing in the Middle Ages: A View from the Trenches 1955-1983*, 1st Books Library.

Parnas, D.L. (1972) On the criteria to be used in decomposing systems into modules , *Communications of the ACM ,* 15(12), 1053–1058.

Raymond, E.S. (1999/2001) *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary,* O'Reilly.

Ritchie, D.M. and K. Thompson (1974) The UNIX time-sharing system, *Communications of the ACM,* 17(7), 365–375.

Salomon, D. (1993) *Assemblers and Loaders.*

Salus, P. (1994) *A Quarter Century of UNIX,* Reading, MA: Addison-Wesley Publishing Company.

Schotchmer, S. (2006) *Innovation and Incentives,* Cambridge, MA: The MIT Press.

Shell, D. (1959) The SHARE 709 system: a cooperative effort, *Journal of the ACM,* 6(2), 123–127.

Stallman, R. (1983) new Unix implementation, a message sent to net.unix-wizards,net.usoft, archived at http://www.gnu.org/gnu/initial-announcement.html, last accessed October 30, 2008.

Stallman, R. (1984/2002) The GNU Manifesto, in J. Gay (ed.), *Free Software, Free Society: Selected Essays of Richard M. Stallman,*Boston, MA: GNU Press, pp. 31-39.

Stallman, R. (1996/2002) Free Software Definition, in J. Gay (ed.), *Free Software, Free Society: Selected Essays of Richard M. Stallman,*Boston, MA: GNU Press, pp. 41-43.

Stein, J. (2007) Conversations among competitors, SSRN Working Papers, http://ssrn.com/abstract=1008328, last accessed on October 30, 2008.

Stern, S. (2004) Do scientists pay to be scientists? *Management Science,* 50(6), 835–853.

Tanenbaum, A. (2004) Some notes on the "Who wrote Linux" kerfuffle, Release 1.5, http://www.cs.vu.nl/~ast/brown/, last accessed October 30, 2008.

Torvalds, L. (2001) *Just for Fun: The Story of an Accidental Revolutionary,* Harper-Collins.

TIOBE (2009) TIOBE programming community index for April 2009, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, last accessed April 22, 2009.

Varian, H. (1994) A solution to the problem of externalities and public goods when agents are well informed, *American Economic Review,* 84, 1278–1293.

von Hippel, E. (1998) Economics of product development by users: the impact of "sticky" local information, *Management Science,* 44(5), 629–644.

von Hippel, E. (2007) Horizontal innovation networks — by and for users, *Industrial and Corporate Change,* 16(2), pp. 293–315.

Weber, S. (2005) *The Success of Open Source,* Cambridge, MA: Harvard University Press.

Williams, Sam. (2002) *Free as in Freedom: Richard Stallman's Crusade for Free Software*, Sebastopol, CA: O'Reilly.

Williamson, O. (1975). *Markets and Hierarchies.* New York: Free Press.

Williamson, O. (1979) Transaction cost economics: the governance of contractual relations, *Journal of Law and Economics,* 22, 233–261.