Network Effects and Software Development – Implications for Security

Jari Råman
Institute for Law and Informatics
Faculty of Law, University of Lapland
P.O. Box 122, FIN-96101 Rovaniemi, Finland
Jari.Raman@urova.fi

Abstract

This survey of literature in software economics tries to find explanations for the vulnerabilities left in commercial software products and the lack of security considerations in software and information systems (IS) that are developed from components. An examination of the information economic environment where market-driven software development occurs, which is characterized by network effects and lock-in, suggests that the schedule and the budget constraints on secure software and IS development are outcomes of rational economic behaviour. The consequences for security of first-to-market competition (e.g. shortening development-cycle, release-oriented development, beta testing) and customer lock-in (e.g. into an insecure product) are considered from an economic point of view.

1. Introduction

The main method of securing IS and software is adding security afterwards; they still are not developed with security in mind [16]. This applies to both aspects of secure software and IS development: the quality aspects of security (reducing vulnerabilities that could pose a risk to security) and the security features that can be seen as requirements of an IS or a software and should be made part of the system according to user needs. The former prevents threats stemming from the system itself (by reducing vulnerabilities in the system) and the latter prevents threats coming from outside the system, e.g. vulnerabilities in the system or persons using it that an outsider accidentally or deliberately abuses.

The quality aspect of security is dominated by the “penetrate-and-patch” approach [44]. Vulnerability avoidance is still overlooked, and security becomes an issue only after a published security breach, even though this practice of waiting until the end of the development cycle to deal with the vulnerabilities has been proved wasteful. After looking at the cost benefits of increased security on application development, Soo Hoo et al. [43] concluded that catching security vulnerabilities in the design phase was more cost effective than in implementation, which is more cost effective than in testing, and so on. Building security (in the sense of fewer vulnerabilities) into applications from the start improves reliability, avoids potentially embarrassing and costly incidents, and ultimately saves money. This kind of attention to quality (customer expectations and requirements) early in the life cycle of a project (at the requirements analysis and specification level) leads to defect detection and avoidance that both increases productivity and improves speed to market [8, 44].

In software engineering, security requirements are seen as complementary to the normal, or functional, requirements of a system (such as the features that the customer would require). The favoured methods for requirements engineering typically do not even include security concerns as an integral part of the process. Although some security concerns are addressed during the requirements engineering stage, most requirements come to light only after functional requirements have been satisfied. As a result, security requirements are added as an afterthought to the standard (functional) requirements [16]. This is the case despite the reasonably wide agreement that security requirements (e.g., confidentiality, integrity and availability) and other security features (e.g., access rights, security classifications) should be defined into the system from the beginning and despite the efforts to integrate security design into the development processes.

Affecting the development of security features even more extensively than this afterthought approach is what may be termed developmental duality. The methods used both in IS and software development have neglected security design, thus requiring separate methods for their secure development [5, 40, 41]. As originally noted by Baskerville [4], the separation of normal IS development from the information security process creates problems such as restrictions on proper system functions, higher costs and shorter system lifespan.
This article tries to explain this contradictory behaviour: security is not taken seriously in the development phase, even though there is evidence of extensive benefits and agreement on the importance of doing so. But qualifications are in order before going into these explanations by looking how network effects contribute to development work.

The approach to software development in this article is that of market-driven (packaged) development – developing for a marketplace (conceivable market segments) rather than for a particular customer or a (group of) user(s) based on contract. This mode of development is used both in commercial off-the-shelf (COTS) and embedded software business models. Other forms of software and IS development are not dealt with, because network effects do not play such an important role in them. This exclusion may be further justified by emphasising the trend to develop and embrace technologies that reduce the amount of new programming (which is difficult, labour-intensive and time-consuming), hence to reduce the costs involved in developing any software or IS. Perhaps the most visible example of the trend to avoid programming functionality from scratch, together with in-house software reuse (reusable software assets developed in-house), is the increased use of (COTS) software components (systems, subsystems, and libraries of components) [36, 12]. These components are used even in safety-critical systems [18].

Another limitation on the scope of this article is that the emphasis of this short probe is mainly on the component developer, unless otherwise stated. What is referred to as market-driven development is thus about the processes of the component developer and not the component user. However, it has to be made clear that a component user is typically also a software system or an IS developer, and the development processes of both sides are affected by the focus on COTS products in the software market. There are even considerations of separate third-party component-based software development processes (see [3]).

2. The network economic environment

A short look at the economics under which market-driven software development operates – network or information economics – is needed in order to be able to understand the reasons why security has typically not been considered in the development phase. Those familiar with software economics issues are welcome to proceed to the following section.

Software as an information good has an unusual cost structure. An information product is typically expensive to produce, but very cheap to reproduce. The fixed costs related to producing the first copy are not only fixed but also largely sunk: fixed costs are not recoverable if production is halted. Also variable costs are typically small: the cost of producing an additional copy typically does not increase, even if a great many copies are made. [39, 33]. Substantial supply-side economies of scale (lower unit costs by being larger) are gained with this ‘high fixed cost of development and low marginal cost of producing subsequent copies’ feature typical in software and information markets (similar to most industries). But demand-side economics of scale (taking advantage of positive relationship between popularity and value) is the norm in information industries. The combination of both demand- and supply-side economies of scale makes the information industries different: growth on the demand side both reduces cost on the supply side and makes the product more attractive to other users; this accelerates the growth in demand even more, resulting in especially strong positive feedback. [39]

Networks of compatible users generate network effects (i.e., the value of the software to the individual user depends on how many other users there are for the same software product) that in turn give rise to positive feedback [39]: as more users deploy the same software, the more communication partners there are to share files and tips with and the more encouragement there is for software houses to devote more resources to developing compatible software. This drives the potential users of software to buy the product they believe will become the dominating one and to keep its position. In this way, they get the most value for their money.

Network markets can be viewed as falling on a continuum that may roughly be divided into actual (direct) networks, virtual (indirect) networks, and simple positive-feedback phenomena [25]. The essential criterion for locating a good along this continuum is the degree to which the good provides inherent value to a consumer apart from any network characteristics. The greater the inherent value of the good relative to any value added by additional consumers, the less significant the network effect. Computer software has been seen as a paradigm example of virtual networks - a good that provides inherent value to consumers that increases with the number of additional users or identical and/or interoperable goods [25]. Virtual network goods need not be linked to a common system, as the constituents of a communications network (actual network) are: very strong positive feedback effects tied to functional compatibility are sufficient. Unlike actual networks (e.g., telephones and

---

1 This is a crude generalisation as the costs are not completely sunk. When software undergoes a series of releases that stem from modifying the existing code and care is put into the design and coding, then less code needs rewriting. Design documentation makes changes easier and if the same people make the modifications, their experience makes future development cheaper. Software engineering methods and processes (such as reuse) are thus intended to allow developers to gain from earlier development work and to make it possible to recover at least some of the cost of developing stable code [9].
fax machines, where the entire value of the product lies in facilitating interactions between users and the benefit to a purchaser is access to other purchasers, goods that constitute virtual networks (e.g., an operating system or an application program) allow even a single user to perform a variety of tasks regardless of other users of the software. However, the value of given software product grows considerably as the number of additional purchasers increases (e.g., easier file sharing, less need for retraining and thus more competent employees available). [25]

In addition to horizontal technological compatibility, software is subject to increasing returns based on positive feedback from the market in the form of complementary goods. Software developers will write more application programs for an operating system with a bigger market share because that operating system will provide the biggest market for applications programs. Conversely, the availability of a broader array of application programs will reinforce the popularity of an operating system. This makes investment in application programs compatible with that system more desirable than investment in programs compatible with less popular systems. Similarly, firms that adopt relatively popular software (not just an operating system) will likely incur lower costs in training employees and will find it easier to hire productive temporary help than will firms using unpopular software. Note that the strength of network effects will vary depending on the type of software in question. Network effects will be materially greater for operating systems software than for applications programs. [25]

One of the most striking consequences of network effects is their impact on the nature of competition between sellers of products embodying different, incompatible standards. When firms compete for a market where there is strong positive feedback, only one will emerge as the winner. As the installed base of users grows, more and more users find adoption worthwhile and the product eventually achieves critical mass (a large enough customer base) and takes over the market. Such markets are called ‘tippy’, meaning that they can tip in favour of one player or another; it is unlikely that all will survive [39]. Whether a market tips or not depends on the balance between economics of scale (on either the demand or supply side of the market) and variety: with strong economics of scale, the market is likely to be tippy, and if different users have highly distinct needs, the market is less likely to tip [39]. As a consequence, growth becomes a strategic imperative, and not just to achieve the usual supply side economics of scale but also to achieve the demand side economics of scale generated by network effects. Obtaining critical mass becomes the key challenge, after which the market is considered to build itself [39]. In its most extreme form, positive feedback can lead to a winner-take-all market, in which a single firm or technology vanquishes all others [39].

This tendency of network markets to tip leads to particularly intense competition early in the market’s existence. As networks take time to build up to critical mass (i.e., become widespread enough to be economically viable), producers that are sufficiently ahead of the competition (in both time and appeal to the market) with a new product or application will be able to acquire the necessary critical mass to exploit economics of scale. Accordingly, the best way to secure market leadership in the presence of the economics of scale typical of information industries is through an early presence in the market: simply being first to market can generate both differentiation and cost advantages [39]. The key is to convert the timing advantage into a more lasting edge by building an installed base of users².

3. Time-to-market and security

Network effects have considerable influence on the behaviour of COTS software producers in particular: a short time-to-market to exploit the first-mover advantage is crucial to establish a software product in the market and to profit from the network effects. Incentives to be the first on the market and to establish one’s own products as de facto standards are very high. Studies in software development repeat the economic rationality that in market-driven development the primary goal is time-to-market [12, 29, 32, 11]. Time-to-market is crucial not only for a new system but also for new product features or concepts in existing systems. Nor does time-to-market constrain only the initial release: market leaders must keep developing advanced features, bug fixes, and performance improvements in order to keep old customers satisfied and to win new ones [34, 6]. In fact, competition for best time-to-market is perpetual.

A shortening development cycle due to time-to-market pressures is a general trend in market-driven development, especially in the development of mass-marketed software (including software embedded in consumer electronics and telecommunications equipment) [14, 8, 24, 9] and Internet software. The development cycles have compressed from 24-36 months to 12-18 months for non-Internet related companies and even to 3-6 months for companies involved in eCommerce and creating and maintaining Web portals [15] (shortening development cycles in Internet software companies have been reported also by MacCormack [27] and MacCormack et al. [28] and in relation to application and smaller niche software developers by Baskerville et al. [6] and by Baskerville and Pries-Heje [7]).

The high incentives to be first to market and to establish one’s own products as de facto standards means that software development practice easily becomes

² Note that being first to market is not necessarily decisive in the long run, even though it usually helps [39].
distorted to maximize functionality and minimize development time, with little attention paid to other qualities (especially non-functional requirements such as security and safety) [36] or project goals that conflict with functionality and time-to-market (fundamentally so in the case of security because more security limits functionality) [44]. Anecdotal evidence shows that companies may compress their quality assurance (QA) practices when the priority is a shorter development cycle [6, 46]. A shorter development cycle results in reduced quality and security, especially if ad hoc processes are used – as they typically are – in the prioritisation and cost/impact assessment of sometimes conflicting requirements such as quality/security costs and time-to-market [34].

For example, despite the wide acknowledgement of the advantages of good requirements specification early in the development phase, the documentation and maintenance of requirements tend to be sketchy. Requirements that are proposed, invented, or designed are communicated within the development organisation to a large extent by word of mouth; they are not elicited or played back to the customer since in market-driven development users (referred to as customers) are typically unknown during the development phase (there may not even be a user until the first release of the product) [36].

Even though there is increasing research on requirements engineering due to the wide recognition of the centrality of requirements specification to the whole development process (e.g. Blackburn et al. [8] and Schneider [36]) and improvements have been made, many of the general challenges in traditional requirements engineering are adopted by the market-driven development organisation (e.g. requirements are erroneous, errors are detected late, and ambiguities are difficult to resolve) [29]. This is often because time pressures change the way requirements are approached. Traditionally, requirements analysis has been based on the assumption that a large company or institution orders a large software system from either an external vendor or internal IS department, using a requirements specification as a contract. However, in market-driven development there is no contractual situation, and requirements specifications are rarely written [12, 29].

An important consequence of the shortening development cycle (due to first-to-market being essential) is the tendency towards release-oriented development. This has been reported for market-driven software development in general [29, 32, 22], as well as software for electronic commerce and for web portals [15, 6]. In interviewing companies using Internet speed development techniques Baskerville and Pries-Heje [7] found that vague requirements continue throughout projects. This is one cause for release orientation.

Constantly striving to be ahead of competitors, the market-driven development company frequently delivers new and improved releases (containing bug fixes and new features) of software products. The development resources – especially time-to-market, but also budget – are typically fixed [1], at the expense of lower-priority requirements. Each feature is examined to determine whether its inclusion in the product is necessary for the product to be competitive in the marketplace. Generally, those features with direct customer appeal win; subtle, hard-to-demonstrate, and pervasive properties – such as security – tend to be rejected [36]. They are put of from one release to another in order to meet the release date. This feature or requirement slippage should not come at the expense of quality and security, even though this easily is the case [42, 6, 12]. Security tends to be ignored because it would require that more time and money be put into development work that is beneficial only in the long run (no immediate returns). Time-to-market considerations discourage the inclusion of security features and encourage the postponement of security to later releases – if they are considered at all.

However, the quality aspect of security is placed high on the agenda when a security problem left in the system is exploited and is being reported. Unfortunately, a patch is essentially the only option used when this occurs. According to Viega and McGraw [44], the problems of this pervasive ‘penetrate-and-patch’ approach are, among other things, that not all problems are reported to the developers that make the patches, that patches often introduce new problems because they are also rushed out as a result of market pressures, that often only the symptom of the problem gets fixed leaving the cause unaddressed, and that patches often go unapplied or are otherwise ineffective.

But it is no surprise that this penetrate-and-patch approach still is so pervasive. Not only does it help in getting products onto the market more quickly by skipping initial security considerations that seem to slow down development, diminish functionality, and give revenue only in the long run, but patches are also cheap to distribute. Distributing a patch for a piece of software is a lot cheaper than for traditional commodities, due to the immaterial nature of the good and consequently the virtually zero marginal cost of transmission. Developers can make patches available to the web or e-mail them out

---

3 The way software-consuming organisations interact with software producers has changed due to the increased product attention. Traditionally, early and close links between users and developers have been considered critical. Today, software consumers and producers use a variety of intermediated means to communicate their needs to developers. For example, packaged software developers build to requirements gleaned from a variety of sources, including help-desk call-log analysis, market research, product reviews, and user groups, of which direct customer contact is one of the least likely means. [35, 23]. This is one of the major dilemmas: the challenges of developing software for larger markets is to satisfy the end user although contact with the end user is limited [29].
to customers and the cost of installing the patch falls on to
customers.

A release orientation is necessary for a start-up company since it cannot generate revenue before it produces functionality, even where this is somewhat unreliable. The code of the first release lacks security because any issues that can be postponed, including quality and security, are disregarded [6]. For an existing (or even dominating) firm, the frequency and timing of new versions and upgrades is a way to control the length of the cycle of customer lock-in into one's products [39]. It is necessary to produce new releases and upgrades to prevent aggregate customer lock-in from getting too low at any point of in time, because the optimal time for a competitor to enter the market and to attack the installed customer base of the existing firm is when the aggregate customer lock-in is low [39].

A fast cycle time together with a release orientation is something that is impossible to achieve in a serial process. Parallel development is used widely in release-oriented development: traditional serial phases are split and assigned to separate groups of developers, which then perform them simultaneously. Quality assurance and testing are also done in parallel with other development phases; if a rapidly approaching release date forces the company to shorten development phases, quality assurance and testing also get short-circuited [6].

4. Remarks on maintenance and testing

The product focus in software development accounts for how software maintenance changes in the software products market. In traditional tailored software development, there usually is just one release and the fixes, additional features and other evaluations are provided as part of maintenance. In market-driven development this work is done by making new releases of the same product [29, 12]. Even though the COTS software vendors separate corrective maintenance – including patches and workarounds which are often provided to licensees at no cost beyond a subscription fee – from other forms of software maintenance, the changes needed to smooth out poorly done but operable software functions become the basis of new releases for which vendors charge additional, often highly profitable licensing fees. In other words, maintenance takes the form of versioning or supporting services, for which the customer has to pay separately. Most of what was once maintenance in traditional software development now forms the basis of a product’s next release and thus serves to generate additional revenue for the vendor over a number of years [35].

The combination of release-oriented development and patching and, especially, the use of new releases as an important form of maintenance lead to a reliance on customer feedback as a significant, or even primary, quality assurance mechanism in market-driven development. This is reasonable economic behaviour as Shneier [37] so effectively puts it in his famous quote “…90% to 95% of all bugs are harmless. They're never discovered by users, and they don't affect performance. It's much cheaper to release buggy software and fix the 5% to 10% of bugs people find and complain about.” (Cf. [18]).

This has implications for quality and security. Press coverage is not guaranteed to be accurate and may not convey the implications of the problem being reported. The problems that concern only a smaller user community do not get fixed. Feedback from customers and the press, by its very nature, occurs only after a product has been distributed. Reliance on market forces to select what gets tested and what gets fixed is haphazard at best and is surely not equivalent to performing a methodical search for vulnerabilities prior to distribution. [36]

The development goal of achieving software ‘good enough’ – not perfect (flawless) – for the specific situation and customer needs also applies to testing. A lot of bugs are detected during testing procedures, but not all errors can be found. As Cem Kaner [21] (among others) explains it, it is impossible to fully test a program: the testing procedure can only show the presence of errors in the program; it cannot show the absence of errors. Additionally, testing cannot show that the software has certain qualities. Despite these limitations, testing is widely used in practice to create confidence in the quality of software [20]. This is why software is shipped with bugs even after the verification and validation stage. However, the bugs that remain do not prevent software from being ‘good enough’.

The verification and validation stage can be one of the more time consuming, expensive and challenging phases of the software life cycle. It has been estimated that about 50% of the development costs of a software product are caused by testing and debugging [20, 19]. In market-driven development (a very competitive market), the design team is often under tremendous pressure to complete this phase. Market pressures contribute to reducing the time spent on testing before releasing software to users. More sophisticated testing and debugging procedures would prolong introduction of a new product (to the market) as well as add costs and thus decrease the probability of commercial success. In sum, software is being released and implemented without adequate testing (similarly in [31]).

There are special problems in testing for security. The testing procedures must be changed to focus on security

---

issues (e.g. testing for unexpected input, probing a system like an attacker or otherwise looking for exploitable weaknesses) in order to find the particular vulnerabilities [31, 44]. Functional testing (treating the component as a black-box and testing the interfaces of the components) does not find security flaws. Unlike almost all other design criteria, security is independent of functionality. Functional testing is good at finding random flaws that, when they happen, will cause the computer program to behave oddly. Security flaws have much less spectacular effects; they are usually invisible unless they fall into the wrong hands. Security testing is not about randomly using the software and seeing if it works, but deliberately searching for problems that compromise security [38, 31, 44]. The costs of testing, together with the time needed, increase when security is concerned, which is why software rarely end up being ‘good enough’ in terms of security even after testing.

The method used in market-driven development – enlisting the user community to help in finding errors by making early releases (beta versions) available to interested users and by freely distributing incremental updates (e.g., patches) to the software – does not enhance security, since no amount of beta testing will uncover security flaws. This is mainly due to the need for very sophisticated attacks in security-specific testing that are not typically widely known in the broader user community. Of the parties searching for vulnerabilities, hackers and professional tiger teams may have the skills and motivation (at least to some point), but are in no way able to do it quickly and efficiently for every software product (on the problems of red teaming for security, see Viega and McGraw [44], pp. 42-43).

A further problem in using customers (either end-users or software developers using components) as testers is that they typically do not have access to the component’s source code and to the specific documentation of the production process, especially where COTS is concerned [18]. COTS vendors seeking to protect their intellectual property usually sell components as binaries, without source code or design documentation. The lack of availability of component source code limits the testing that the component user can perform (white-box techniques in evaluation of components is not possible). Even though some traditional security analysis is made impossible for the component user or other customer by the absence of source code, there are ways for the user to verify and determine the quality and security of COTS components that do not require extensive disclosure of the source code or the accommodated design documentation. There are approaches that treat the component as a black box, and employ extensive testing to ensure that the system functions as desired; no additional effort or disclosure of intellectual property rights (IPR) are required from the COTS vendor [45]. Grey-box verification systems use interactive cryptographic techniques or rely on tamper-resistant hardware to help the vendor to provide evidence of the quality and security of the component (disclosure of enough details of the verification practice to convince a sceptical component user) without disclosing too much information that could endanger its IPR [16]. There are also different sets of criteria on which components are evaluated and verified (e.g. ITSEC, TCSEC, Common Criteria).

Even though the additional testing effort required by black-box approaches contributes towards the overall quality of the component user’s entire system, their use is limited because the additional testing is likely to be time-consuming and expensive. An additional limitation on the use of black-box approaches is that they do not reveal unknown, malicious functionality [45]. Grey-box approaches have only very recently appeared and need a lot of additional research. However, with additional research into the ways in which component users can test systems efficient techniques and tools are likely to emerge that will help such users test their applications more effectively.

The use of customer feedback in place of other quality control mechanisms does allow a software producer to externalise costs associated with product testing. Customers, in turn, have to invest time and money in finding and possibly reporting errors, in installing patches, and they also have to suffer from the costs of failures. The tactic of using customers as serious (perhaps involuntary) testers is, at best, a dubious one from the point of view of security [44].

5. Appeal to developers and security

When competing to dominate the network market, i.e. to achieve the critical mass needed to take it over, firms have to appeal strongly to the developers of the next-generation ‘killer-aps’ and vendors of complementary goods and services [2, 1, 30]. Good developers are needed to create the products that attract customers in the short timeframe required and vendors of complementary goods help to build up the critical mass of users for the product and its applications. Also the appeal to early adopters has to be strong since they are the critical mass – if they are pleased. One can worry about the rest of the end-users later.

This helps to explain why the security features in commercial software, if they are even implemented to begin with, are made easy to bypass. Stronger security that is not easy to bypass may diminish functionality and require more work from and less opportunities (increased costs and diminished revenues) for the developers of the next generation ‘killer-aps’ and developers of complementary goods and services. Since early adopters typically are persons with high technology skills that want...
to try out and test new things, reduced functionality and security that is not easily bypassed might lower the appeal of the product among this conceivable critical mass. Due to the relative unimportance of end users (they just want to get a few crucial tasks done) and the crucial role of the developers who create the tools that attract the early adopters, the cost of insecurity (caused, e.g., by vulnerabilities being abused and when upgrading the system) and other support costs needed to operate with the software (from implementation, maintenance, testing and dealing with the failures) is dumped on to the end-users. These costs have long been hidden. [2, 1]

In an attempt to appeal to the early adopters and thus to exploit the positive feedback, vendors may add features to attract this small number of users with special needs even if those features will be unused by other users and result in lower security (due to an increase in the number of vulnerabilities, the usual lack of testing in these unused parts, and the security problems incurred when integrating these additional features into the system).

Because consumers typically possess imperfect information about the product’s security, vendors can get away with both of these practices.

6. Security and lock-in

With network effects the costs of coordinating a large group of individuals to switch to a competing product can be extremely large; the effects contribute to customer lock-in (i.e., the costs of switching from a product to another are so large that switching suppliers is virtually unthinkable). And because the building of network size by a competitor with an incompatible product requires overcoming the collective switching costs – the combined switching costs of all users – customer lock-in becomes the norm in the information economy [39]. If an installed base of users is established before the competition arrives on the scene, achievement of the scale economics necessary to compete can be made difficult for later entrants [39]. Thus, first-mover advantage is not only powerful but it can also be long lasting in lock-in markets.

One implication of the first-mover advantage together with lock-in in software markets is that the market may settle on a good with a lower social valuation. Once the market tips toward a single standard, it may remain on that standard and its successors for a long time even though an objectively ‘better’ standard is available. Even though all users would be better off with the new standard, those benefits do not accrue to the present users, who would have to pay substantial switching costs. New purchasers also may opt for the established standard because of the immediate benefit that the established network offers; they do not take account of the benefit that purchasing the new product would confer on later purchasers. Even if they anticipate that the new product will be widely adopted, the benefits of that adoption to new purchasers may be realised so far in the future that they are substantially discounted.

In the present context, this theoretical possibility means that the market may get locked-in into a product that is insecure (has security related vulnerabilities or lacks security features), even though a more secure product is available in the marketplace. This is because users do not, in network markets, choose software purely on the basis of its features and security. Network effects are also significant, because the utility of software product increases with the number of users and users of compatible products. A typical user may need a certain set of features and beyond that be concerned with standardisation and interoperability. When a market has tipped in favour of one product, users who, in autarchy, would be willing to sacrifice functionality for security may choose a less secure product because of the benefits of interoperability. Because of network effects, users who would otherwise prefer increased security to increased functionality might then choose less secure but more widely used programs. There is no expectation in favour of markets getting locked into an inferior (less secure) product, but it is a possibility that has to be taken seriously: when it actually occurs, it has non-trivial consequences, and there exists at least one other feasible state of affairs that might be preferable [26].

In the case of market-driven development, lock-in into a vulnerable product is likely. As discussed above, with first-to-market as an essential feature in achieving a market share needed for bearing the competition, security tends to become an afterthought. However, the winner ought to make its product better in quality and security over time. Yet this might take a long time or not happen at all, because the code of the first versions has emphasised functionality over security and a complete rewrite of it is not practical given the evolutionary model of development: development consists of modifying previous versions and, over the years, these become so complex that they simply could not be developed (or redeveloped) from scratch [10] (Cf. Baskerville et al. [6] p. 53). Thus, the design space of release-oriented development is highly influenced by the state of the code after the previous release(s). Because adding security later in the development phase is difficult and expensive, and when previous release(s) has (have) emphasized functionality over security, increasing the security of it in a new release

---

5 Network effects are a common source of switching costs: when a product has become ubiquitous it is very costly to switch to something new [39]. Internet distribution of new applications and standards reduces some of the network effects for software by reducing switching costs. Variety can also be supported more easily if an entire system can be offered on demand. However, the Internet does not eliminate network effects in software; interoperability is still a big issue on the supply side and there still is a strong need for standardisation. [39]
requires significant resources that will not then be available for enhancing functionality. But, since functionality still has to be improved in order to make the new release attractive (due to heavy competition), the resources are easily taken from features that are considered less useful (e.g. quality and security). This means that initially insecure code does not necessarily improve with new releases.

The positive aspect of lock-in is that software and content producers in general have an increased interest in security mechanisms for their value in locking customers into their products and systems [26, 2]. This can be achieved by using information security protocols to set technical compatibility requirements that must be met by connected applications. A supplier wishing to sell a system component will either need to be compatible with the necessary security protocols (by licensing or reverse engineering), or must provide sufficient added value to motivate a customer to replace all other system components that require those security protocols, which may result in prohibitive switching costs.

The strong and effective lobbyists of the rights holders and software vendors thus have incentive to try to have laws enacted that protect their commercial interests (from illegal copying) and, at the same time, silently enhance customer lock-in; all one has to do is look at the anti-circumvention rules in the EC Copyright and Conditional Access Directives and the US Digital Millennium Copyright Act. When security is brought up, the rights holders worry about the safety of their respective intellectual property assets and possibilities to keep their market share but do not care much about the security of the underlying information infrastructure. This attitude poses a critical problem for security in a networked world: laws that protect possibly insecure software are going to protect network insecurity at the same time [17].

7. Conclusion

The reason why software quality is so low compared to other practice (engineering) and research fields, and why software and IS still are not developed with security in mind is partly explained by the immaturity of software engineering and information systems disciplines: the methods are constantly evolving (novel methods arise every now and then, and are modified by practitioners to fit different situations), and they do not consider information security design issues, which are relegated to separate secure IS and software development (developmental duality).

While the state of methods for development is still rather inadequate, there are well-known technical and procedural ways to prevent at least the widely known vulnerabilities and methods to integrate security in development processes. Thus, given enough demand from customers, vendors could supply improved quality and security. It is not that developers are incapable of producing software with fewer vulnerabilities or including security features into systems; it is just that they are not sufficiently motivated to do so. Beyond the intrinsic difficulty of writing defect-free software, there are constraints on the development of secure software that result from the business environment: the constraints for the COTS software business model largely derive from schedule and budget. The economic considerations imply that no amount of technology or methodological improvement (even acknowledging the need for and importance of further work on them) is able to give satisfactory solutions to security; the presence of network effects as such is enough to pose serious economic disincentives. Fortunately, their seriousness have been recognised and efforts have been made for their correction.

The sub-optimal equilibrium, in which a vendor can be successful without improving security, continues as long as customers accept the situation, the primary demand for functionality only (the main purpose of a computing or a communication device or a system) continues to grow and fuel demand for features [36], and developers do not consider the current release’s possible influence on options in future releases. The implicit argument is (basing its claim on welfare economics) that changes in the private motivation to provide secure enough software and IS can lead to a more favourable equilibrium; network effects do not prevent this unless the participants in the market fail to internalise them – and even then the incurred ‘network externalities’ are not likely to have serious consequences on security. The market actors themselves are enough to provide security.

Unfortunately, this is not expected to happen in the near future. Customer and end-user expectations seem to be fulfilled due to the acceptance of a reasonable degree of insecurity and unreliability or operational difficulty as a trade-off for innovation. Yet, this tolerance of poor quality and lack of security will vanish with time (perception of product as less innovative) and competition (competitors may introduce new products and releases with new features, and users will very quickly lower their tolerance if a competing product does better). But the dominant players can delay the effect of competition in particular, because high switching costs cause customer lock-in into current products.

But even if the private motivation to enhance security amends, it might not suffice to provide an optimal amount of security for society as a whole. However, whether or not a government intervention is needed in the software product market merit a separate study bringing in considerations of welfare economics.
9. References


Oulu, Finland, June 22-24, 1999, Espoo: Technical Research Center of Finland.


