
How to Evaluate Open Source Software / Free Software (OSS/FS) Programs

David A. Wheeler

dwheeler@dwheeler.com

Revised as of November 21, 2003

This paper describes a general process for evaluating programs, with specific information on how to evaluate Open Source Software / Free Software (OSS/FS) programs. This process is designed so that you can compare OSS/FS programs side-by-side with proprietary programs and other OSS/FS programs, and determine which one (if any) best meets your needs. This process is based on four steps: [identify candidates](#), [read existing reviews](#), [briefly compare the leading programs' attributes to your needs](#), and then [perform an in-depth analysis of the top candidates](#). Important attributes to consider include [functionality](#), [cost](#), [market share](#), [support](#), [maintenance](#), [reliability](#), [performance](#), [scaleability](#), [useability](#), [security](#), [flexibility](#), and [legal/license issues](#).

[1. Introduction](#)

[2. Identify candidates](#)

[3. Read existing reviews](#)

[4. Briefly compare the leading programs' attributes to your needs](#)

[4.1 Functionality](#) * [4.2 Cost](#) * [4.3 Market Share](#) * [4.4 Support](#) * [4.5 Maintenance/Longevity](#) * [4.6 Reliability](#) * [4.7 Performance](#) * [4.8 Scaleability](#) * [4.9 Useability](#) * [4.10 Security](#) * [4.11 Flexibility](#) * [4.12 Legal/license issues](#) * [4.13 Other Issues](#)

[5. Perform an in-depth analysis of the top candidates](#)

[5.1 In-depth Analysis for Adding Functionality](#) * [5.2 In-depth Analysis of Software Security](#)

[6. Wrap-up](#)

1. Introduction

[Open Source Software / Free Software \(OSS/FS\)](#) has risen to great prominence. Briefly, OSS/FS programs are programs whose licenses give users the freedom to run the program for any purpose, to study and modify the program, and to redistribute copies of either the original or modified program (without having to pay royalties to previous developers). [Many quantitative studies have shown that, in many cases, using OSS/FS programs is a reasonable or even superior approach compared to their proprietary competition.](#) OSS/FS programs are also called FLOSS, FOSS, and libre programs.

This paper describes how to evaluate programs, including OSS/FS programs, to determine which one (if any) best meets your needs. This paper is intended for those who already know how to evaluate proprietary programs, but are not sure how to evaluate OSS/FS programs. Significant technical mastery of software development isn't required for most of the process described here, but technical knowledge about software development is certainly helpful and there are a few (marked) steps which are best performed by someone knowledgeable about software development. For those who want more details, this paper includes many links to supporting material.

The basic steps for evaluating all programs, both OSS/FS and proprietary, are essentially the same. I suggest the following steps: *identify* candidates, read existing *reviews*, briefly *compare* the leading programs' attributes to your needs, and then perform an in-depth *analysis* of the top candidates. You can think of this as "IRCA": [identify](#), [review](#), [compare](#), and [analyze](#). The rest of this paper is, in fact, organized by these four steps, followed by a [wrap-up](#).



However, the way that you perform these steps in an evaluation is different for OSS/FS programs than for proprietary programs. A key difference for evaluation is that the information available for OSS/FS programs is usually different than for proprietary programs. Most OSS/FS programs have a great deal of publicly available information that isn't available for proprietary programs: the program's source code, analysis by others of the program design, discussions between developers about its design and future directions, discussions between users and developers on how well it's working (or not), and so on. An even more fundamental difference between OSS/FS and proprietary programs is that OSS/FS programs can be changed and redistributed by customers. This difference affects many factors, such as [flexibility](#), [support options](#), and [costs](#).

If you're comparing proprietary programs with OSS/FS programs, you can use the general approach described here. You can use the detailed evaluation processes you already know for proprietary programs, and use the details in this paper to evaluate the OSS/FS programs. Details on how to evaluate proprietary software are out of scope of this paper, since I presume you already know how to do that. Some organizations may decide that they wish to only use OSS/FS programs. However, even in that case, you still need to be able to evaluate OSS/FS programs, because you will always need to know how well a given program meets your needs and there are often competing OSS/FS programs.

The amount of effort you should spend evaluating software is strongly dependent on how complex and important the software is to you. The whole evaluation process might take 5 minutes for a small program, or many months when considering a mammoth change to a major enterprise. The general process is the same; what is different is the amount of rigor (and thus effort) in each step.

This paper presumes you have a basic idea of what you need. If you don't, you'll need to first determine what your basic needs are. Usually you will refine your understanding of what your needs are as you evaluate, since you're likely to learn of capabilities you hadn't

considered before. However, you cannot reasonably evaluate products if you don't know what you want them to do for you.

2. Identify candidates

The first step is to find out what your options are. You should use a combination of techniques to make sure you don't miss something important.

An obvious way is to ask friends and co-workers, particularly if they also need or have used such a program. If they have experience with it, ask for their critique; this will be useful as input for the next step, obtaining [reviews](#).

One way is to use a good Internet search engine, and search for the kind of product you're looking for. One good search engine is [Google](#); other good search engines include [Teoma](#), [Alltheweb](#), and [AltaVista](#). Try to avoid search engines with obvious conflicts of interest, e.g., a search engine owned by a maker of one product may not help you learn about their competitors. Some search engines (like Google) accept payment but place paid results separately from the unpaid results - this is fine, and the paid articles can certainly help you identify options, but be sure to review the top unpaid articles too. You should try several variations on the name of the kind of program you're looking for. If you know the name of an existing well-known product, you might search for that name plus words like "compete" or "competitor" to find its competition. Once you know the names of several products, search for the combination of names so you can find pages that list the products of that type.

If there's a naming convention for the kind of program you're looking for, exploit that convention while searching. For example, programs that translate one data format into another often follow the naming convention "x2y", where x and y are the filename extensions. Thus, "gif2png" is a likely name for a program to convert GIF to PNG. Also try "to" instead of "2", and if that doesn't work, search for alternative data formats you can easily convert to (e.g., try "rtf" if "doc" doesn't work).

Another approach is to search on specialized sites which try to track OSS/FS programs. [Freshmeat](#) has a lengthy list. [Icewalkers](#) maintains a list, but note that it only tracks programs that run on Unix/Linux. The Free Software Foundation's "[Free Software Directory](#)" is somewhat smaller, but they work hard to make sure their information is accurate (in particular, they check licenses carefully).

You can also go to sites which host or include many OSS/FS projects. Search sites that host many OSS/FS projects such as [SourceForge](#) and [Savannah](#). Look at Linux distributions and see what they include. [Debian](#) includes an especially large set of OSS/FS projects in its distribution, for example, because since they are Internet-based it's easy for them to include a package for nearly any project.

Finally, look at lists of OSS/FS programs, including any list of "generally recognized as mature" (GRAM) or "generally recognized as safe" (GRAS) OSS/FS programs. After all, some OSS/FS products are so well-known that it would be a terrible mistake to not consider them. For example, anyone who needed a web server and failed to at least consider Apache would be making a terrible mistake; Apache is the market leader and is extremely capable. Here are a few such lists:

1. [My OSS/FS Generally Recognized as Mature \(GRAM\) list](#).
2. The Interchange of Data between Administrations (IDA) programme is managed by the European Commission, with a mission to "coordinate the establishment of Trans-European telematic networks between administrations." IDA has developed [The IDA Open Source Migration Guidelines](#) to describe how to migrate from proprietary programs to OSS/FS programs. This paper includes a list of suggested OSS/FS programs, emphasizing mature products.
3. [The table of equivalents / replacements / analogs of Windows software in Linux](#) lists "equivalent" OSS/FS programs to common proprietary programs. Note that not all OSS/FS programs in this table of equivalents/ replacements/ analogs are mature, and that not all programs in the table are OSS/FS.

3. Read existing reviews

After you've identified your options, read existing evaluations about the alternatives. It's far more efficient to first learn about a program's strengths and weaknesses from a few reviews than to try to discern that information just from project websites.

The simplest way to find these reviews is to use a search engine (like [Google](#)) and search for an article containing the names of all the candidates you've identified. Also, search for web sites that try to cover that market or functional area (e.g., by searching for the general name of that type of product, as you should have already done), and see if they've published reviews. In the process, you may even identify plausible candidates you missed earlier.

It's critical to remember that many evaluations are biased or not particularly relevant to your circumstance. Systems that allow multiple people to comment (like [Freshmeat's](#) "rating" value) can be easily biased by someone intent on biasing them. Still, it's worth hearing a few opinions from multiple sources. In particular, evaluations often identify important information about the programs that you might not have noticed otherwise.

An important though indirect "review" of a product is the product's popularity, also known as market share. Generally you should always try to include the most popular products in any evaluation. Products with large market share are likely to be sufficient for many needs, are often easier to support and interoperate, and so on. OSS/FS projects are easier to sustain once they have many users; many developers are originally users, so if a small percentage of users become developers, having more users often translates into having more developers. Also, developers do not want their work wasted, so they will

want to work with projects perceived to be successful. Conversely, a product rapidly losing market share has a greater risk, because presumably people are leaving it for a reason (be sure to consider whatever its replacement is!).

Market share is extremely hard to measure for most OSS/FS products, because anyone can just download and install them without registering with anyone. However, [market share data is available for some common products \(such as operating systems and web browsers\)](#). This is especially possible with programs that provide Internet services, because programs can be used to sample the Internet to see what's running. Download counts and "popularity" values (e.g., from Freshmeat and SourceForge) can also hint at market share, but again these are easy to bias. Just searching for references to the program name are usually misleading, since many names aren't unique to a particular project. For OSS/FS projects, a partial proxy for market share is how often people link to its project page. Web search engines can often tell you how many links there are to a given project home page (under Google, select Advanced search and then use "find pages that link to the page"). A "link popularity" contest can at least suggest which OSS/FS project is more popular than others. Note that link popularity may only show widespread interest (e.g., it's an interesting project), not that the product is widely used or ready for use.

An interesting indirect measure of a product is whether or not it's included in "picky" Linux distributions. Some distributions, such as Red Hat Linux, intentionally try to keep the number of components low to reduce the number of CD-ROMs in their distribution, and evaluate products first to see which ones to include. Thus, if the product is included, it's likely to be one of the best OSS/FS products available.

4. Briefly compare the leading programs' attributes to your needs

Once you've read other reviews and identified the leading OSS/FS contenders, you can begin to briefly examine them to see which best meet your needs. The goal is to winnow down the list of realistic alternatives to a few "most likely" candidates. Note that you need to do this *after* reading a few reviews, because the reviews may have identified some important attributes you might have forgotten or not realized were important. This doesn't need to be a lengthy process; you can often quickly eliminate all but a few candidates.

The first step is to find the OSS/FS project's web site. Practically every OSS/FS project has a project web site; by this point you should have addresses of those web sites, but if not, a search engine should easily find them. An OSS/FS project's web site doesn't just provide a copy of its OSS/FS program; it also provides a wealth of information that you can use to evaluate the program it's created. For example, project web sites typically host a brief description of the project, a Frequently Asked Questions (FAQ) list, project documentation, web links to related/competing projects, mailing lists for developers and

users to discuss the program or project, and so on. The [Software Release Practice HOWTO](#) includes guidance to developers on how to create a project web site; the [Free Software Project Management HOWTO](#) provides guidance to those who manage such projects.

In rare cases there may be a "fork", that is, competing projects whose programs that are based on a single original program. This sometimes happens if, for example, there is a major disagreement over technical or project direction. If both projects seem viable, evaluate the forks as separate projects.

Next, you can evaluate the project and its program on a number of important attributes. Important attributes include [functionality](#), [cost](#), [market share](#), [support](#), [maintenance](#), [reliability](#), [performance](#), [scaleability](#), [useability](#), [security](#), [flexibility](#), and [legal/license issues](#). The benefits, drawbacks, and risks of using a program can be determined from examining these attributes. The attributes are the same as with proprietary software, of course, but the way you should evaluate them with OSS/FS is often different. In particular, because the project and code is completely exposed to the world, you can (and should!) take advantage of this information during evaluation. Each of these will be discussed below; if there are other attributes that are important to you, by all means examine those too.

4.1 Functionality

One of the most important questions is also the simplest: Does the program do what you want it to do? It's often useful to write down at least a brief list of the functions that are important to you. Many project web sites provide a brief, easily-accessible description of the current capabilities of their program; you'll want to look at this first. For more information, you can usually read the project's Frequently Asked Questions (FAQ) list and the program documentation.

The specific functions you need obviously depend on the kind of program and your specific needs. However, there are also some general functional issues that apply to all programs.

In particular, you should consider how well it integrates and is compatible with existing components you have. If there are relevant standards (de jure or de facto), does the program support them? If you exchange data with others using them, how well does it do so? For example, [MOXIE: Microsoft Office - Linux Interoperability Experiment](#) downloaded a set of representative files in Microsoft Office format, and then compared how well different programs handled them.

You should also consider what hardware, operating systems, and related programs it requires - will they be acceptable for you (do you have them or are you willing to get them)? The issue of operating system requirements is particularly important for organizations that only have Microsoft Windows systems. This is because many OSS/FS programs are only available for GNU/Linux or Unix. In some cases the program can be

quickly ported to Windows, but this often doesn't work as well because Windows does not support some important Unix/Linux features (such as the low-level "fork" capability), or Windows works significantly differently (e.g., its graphical user interface). Sometimes, particularly with server applications, it may be much better to get an inexpensive computer and install GNU/Linux, FreeBSD, or OpenBSD to use an OSS/FS program than trying to port it to Windows. Today's computers and OSS/FS operating systems are so inexpensive that it's often cheaper to buy special-purpose computers for a task than to try to change the application to run on a different operating system. A positive side-effect is that using a special-purpose computer usually improves security significantly, because you can remove all services from the computer that aren't necessary for its specific task. You can often control server applications using web browsers or remote terminals, so in most cases you can have different operating systems for the desktop and the server application.

Few programs (proprietary or OSS/FS) provide *all* functionality you would like. It's often possible to decide to do without, to supplement the missing function with some procedure, or use a separate program to supplement the missing functionality. Some programs have mechanisms that partly give them them additional flexibility - see the discussion on [flexibility](#).

One additional option essentially unique to OSS/FS is that you can have the missing functionality added to the program itself, by changing its code. An organization can add functionality by developing the changes in-house, or by paying someone - such as the project leads - to add the functionality. Many people have found adding functionality to OSS/FS projects is well worth it. Indeed, OSS/FS projects succeed by operating as consortias, where people pool their modifications into a common project. Adding functionality to an existing program is usually far less costly than building the program from scratch, and usually the maintenance costs can be borne entirely or almost entirely by the project instead of the original developer. However, adding functionality increases the cost of the OSS/FS program, there is a time delay before the new functions are ready, there's always the risk that the addition will not happen or work as planned, and there's also the risk that the changes will not be accepted into the program project (resulting in support and maintenance problems). If adding such functionality is important, discuss these new functions with the project developers (to reduce the risks) and account for the cost and time of doing so. Yet do not ignore the option of adding functionality; the ability to add functionality is a key differentiator of OSS/FS, and organizations will miss important opportunities if they fail to consider this option.

4.2 Cost

Clearly cost is an important issue for anyone. Strictly speaking, most OSS/FS programs don't cost anything to get, or have a relatively nominal acquisition cost (e.g., a fee for a boxed CD-ROM set). However, the term "free" in the phrase "free software" is based on "freedom" and not on price. OSS/FS programs still cost money to deploy in the real world, because initial licensing costs are a minority of the costs in most software deployments.

Thus, when considering costs, you should consider all costs related to deploying a program. This is typically done by computing the total cost of ownership (all costs related to deploying the program over a period of time) or as a return on investment (by comparing the total costs to the total benefits), over a fixed period of time.

Thus, for each option you're considering, you should consider all costs, such as initial license fees, installation costs, training costs, support/maintenance costs, license upgrade fees (usually nominal for OSS/FS programs), transition costs (such as data transition and/or transitions to upgrades), and the costs of any necessary hardware. You may want to separate one-time costs from continuing costs, so that one-time costs such as transition costs do not have an excessive weight.

Many proprietary vendors correctly complain that customers too often only look at the initial price. However, proprietary vendors often have "hidden" costs as well, so this complaint applies to both proprietary and OSS/FS programs. Certainly, be sure to include all costs in your calculations, not just the cost of buying initial licenses. Other costs may include expensive hardware (new hardware or upgrades), other software that is required but sold separately, support fees, and any separate upgrade fees. It would be wise to consider this likely if it will be expensive to change vendors later or if the vendor has a history of doing this. One risk factor is the extent a product "locks in" users through proprietary protocols and formats; if the protocols and formats are undocumented, non-standard, or require fees to implement, there is a greater risk of later hidden fees from that vendor since these interoperability barriers could be exploited by the vendor.

Do not ignore - but do not overestimate - training costs. While training costs are important, often products that perform similar functions will have similar training costs. Training costs are rarely a product differentiator, but occasionally a product is so much easier to use that it has a strikingly lower training cost. Transitioning data can be costly, but often people who understand how to use one system can quickly adapt to use another; as a result, the costs of retraining people can be easily overestimated. Organizations should certainly consider paying for support for OSS/FS products, but since OSS/FS support can be competed or self-supported, in many cases this is less expensive than proprietary vendors. Again, you will need to examine your specific circumstances.

4.3 Market Share

As I noted earlier, it's important to know how popular an OSS/FS program is (at least, compared to other OSS/FS programs). The previous section described how to find this out, however, when you report the attributes of each program, make sure you include this information.

4.4 Support

For purposes of this paper, the term "support" covers several areas: training users on how to use the product, installing the product, and answering users who have specific

problems trying to use a working product (including suggesting work-arounds for weaknesses in the current product). For more about issues involving fixing the product and adding new capabilities, see the [maintenance](#) section below.

One major difference between OSS/FS and proprietary programs is how support is handled. Fundamentally, OSS/FS program users have several choices: (1) they can choose a traditional commercial support model, where they pay someone (typically a company) to provide support, (2) they can choose to provide support in-house (designating some person or group to do the support), or (3) they can depend on the development and user community for support (e.g., through mailing lists). These choices apply to each of the various tasks (training, installing, answering questions, fixing, adding new capabilities), and the answers can even be different for the different tasks.

Choosing a traditional commercial support model adds another nuance: which company? Unlike proprietary support (which is usually only provided by the proprietary vendor), there may be several competing companies offering support. These should be evaluated in the usual way: looking at the company's reputations, consider the company's financial health, talking with their existing customers, and so on. But in addition, you should favor companies that are clearly directly involved developing the project's software. Organizations that include developers clearly demonstrate an understanding of the software, can potentially fix any problems that you have with it, and can have the fix incorporated into the main project so that you won't have that problem again in the future. Look at the OSS/FS project page and see which organizations are contributing code to the project; this is usually easy to determine from the email addresses of contributors. In many cases there is a single company who offers primary support for a project in a manner similar to proprietary vendors, typically employing a key project developer; that company is usually the best choice for commercial support, because they will know the project direction and can quickly respond to your needs. Many organizations have ongoing relationships with suppliers and consultants who are qualified to do at least some of these tasks.

Some organizations - particularly large ones which will fundamentally depend on the given product - may choose to provide "organic" support (i.e., create their own support organization for the product). This is probably better considered after having used the product for a while; it can be risky to do self-support without having already had significant in-house experience with the product.

It's even possible to depend on the development and user community for support. This is not as insane as it sounds; in 1997 InfoWorld awarded the "Best Technical Support" award to the "Linux User Community," beating all proprietary software vendors' technical support for the year. However, this doesn't mean that all OSS/FS products are well-supported this way, or that all users should choose this route. This is often the least expensive option, and for extremely cash-strapped organizations it may be the only option. However, this option is probably better considered only after having used the product for a while; it can be risky to depend solely on community support without having already had significant in-house experience with the product (or at least with the

community's support). To evaluate this option, look at the archives of the mailing list(s) used for customer questions, and see how often (and well) customer questions are answered to the customer's satisfaction. It's not possible to answer all possible questions, but you should see honest attempts and successes occurring in most situations. It's important to note that questions can usually only be answered if the questioner gives enough information, so only count questions where the questioner actually does so (see ["How to ask questions the smart way"](#) for what a question should look like, and only consider those questions when examining a mailing list's responsiveness).

4.5 Maintenance/Longevity

Few useful programs are completely static. Needs change, new uses are continuously created, and no program of any kind is perfect. It's important that a program is being maintained, and that it will be maintained far into the future. Of course, predicting the future is very difficult. However, if a program is being actively maintained, it's far more likely that the program you choose today will be useful tomorrow.

OSS/FS maintenance options are essentially the same as those for support, and in reality maintenance and support are not completely separate. As noted above, if your OSS/FS support organization has experience in maintaining the product, they'll be able to make changes or fixes as necessary.

The OSS/FS project site serves as a focal point collecting the improvements of various users. Thus, the project's Internet presence can give you an indication of how well the program is maintained. Examine the developer mailing list archives - is there evidence they're actively discussing improvements to the software? Are there multiple developers (so that if one is lost, the project will easily continue)? If their version management information is accessible to the public, take a look - are developers regularly checking in improvements and bug fixes? [Freshmeat](#) has a "vitality" measure that may help. Some OSS/FS programs are posted and then never maintained; depending on such programs is far riskier than depending on OSS/FS programs that are actively maintained. In general, is there evidence that the software is under continuous development, or has work halted?

Of course, for relatively focused applications, a lack of changes could indicate that the program "just works". This could be suggested by few maintenance changes, but a large number of users and generally positive reviews.

Of course, if a valid legal action is likely to stop the project, then that would critically harm its longevity. This is extremely rare, since developers' actions and code are exposed for all the world to see. For more information, see the section on legal issues.

And to be fair, do not make the mistake assuming that proprietary software has automatic advantages in longevity. In fact, one of the significant advantages of OSS/FS is that you can use the source code to self-support or reconstitute (with other users) a support project. This is in stark contrast to proprietary programs, whose support can disappear suddenly. For example, [the company Appgen, maker of a proprietary business accounting](#)

[package disappeared](#), completely stranding all its users (and the value-added resellers, who had paid \$2,000 or more each for developer kits). In this case, there were contracts promising that the source code would be held in escrow, but users have been unable to acquire the source code, and now they can't even reinstall the software. The article notes that Microsoft's Steve Ballmer, under oath, threatened to stop selling (and supporting) Windows if the Microsoft antitrust trial had produced a final verdict the company found unacceptable. It's unclear if he would have actually done this, but it is clear that the company could have survived a long time without selling a product (by living on its cash on hand) and caused great pain to its customers. A company need not go out of business; if a proprietary product's company decides that the product is no longer in its interests (or they require an upgrade incompatible with your needs), there is no real recourse.

4.6 Reliability

Reliability is difficult to measure, and strongly depends on how the program is used. Problem reports are not necessarily a sign of poor reliability - people often complain about highly reliable programs, because their high reliability often leads both customers and engineers to extremely high expectations. Indeed, the best way to measure reliability is to try it on a "real" work load, as discussed later.

Still, information is often available that may help gauge a program's likely reliability. In particular, a mature program is far more likely to be reliable. The project's web site itself is likely to try to describe the program's maturity; if the project declares that the program is not ready for end-users, they're usually right. To be fair, some developers are perfectionists and are never willing to admit that a program really is mature. [Freshmeat](#) includes a maturity measure that may be helpful. [SourceForge](#) includes a "Development Status" measure for projects it hosts. I have tried to identify some [Generally Recognized as Mature \(GRAM\) OSS/FS Programs](#); feel free to see that list. OSS/FS programs that are included in many GNU/Linux distributions are much more likely to be mature, since distributors don't want to receive a torrent of help requests (and immature products are likely to generate that torrent).

4.7 Performance

Many project websites include performance data. Some OSS/FS projects, unsurprisingly, only present the most positive performance data near their front pages, so this may not present a full picture. (To be fair, that's true for proprietary vendors too.) Project mailing lists may include more detailed performance information. The best way to measure performance is to try it on a "real" work load specific to your circumstance, as discussed later.

4.8 Scaleability

Scaleability, in this context, suggests the size of data or problem the program can handle. If you expect the program to be able to handle unusually large datasets, or be able to

execute on massively parallel or distributed computers, there should be some evidence that the program has been used that way before.

4.9 Useability

Useability measures the quality of the human-machine interface for its intended user. A highly useable program is easier to learn and easier to use.

Some programs (typically computer libraries) are intended only for use by other programs, and not directly by users at all. In that case, it will typically have an application programmer interface (API) and you should measure how easily programmers can use it. Generally, an API should make the simple things simple, and the hard things possible. One way to get a measure of this is to look for sample fragments of code that use the API, to see how easy it is to use.

For applications intended for direct use by users, there are basically two kinds of human-machine interfaces used by most of today's programs: a command-line interface and a graphical user interface (GUI). These kinds are not mutually exclusive; many programs have both. Command line interfaces are easier to control using programs (e.g., using scripts), so many programmers and system administrators prefer applications that have a command line interface available. So, if the application will need to be controlled by programs sometimes, it is a significant advantage if it has a command line interface. There are alternative user interfaces for special purposes. For example, there are programs (typically older programs) which use character screens to create a text-based GUI-like interface (these are sometimes called "curses" programs, named after a library often used to build such programs). However, for most applications intended for direct use by humans, today's users want a GUI; GUIs are much easier to use for most users. In most circumstances, it will be the program's GUI that you'll be evaluating.

It's important to note that many OSS/FS programs are intentionally designed into at least two parts: an "engine" that does the work, and a GUI that lets users control the engine through a familiar point and click interface. This division into two parts is considered an excellent design approach; it generally improves reliability, and generally makes it easier to enhance one part. Sometimes these parts are even divided into separate projects: The "engine" creators may provide a simple command line interface, but most users are supposed to use one of the available GUIs available from another project. Thus, it can be misleading if you are looking at an OSS/FS project that only creates the engine - be sure to include the project that manages the GUI, if that happens to be a separate sister project.

In many cases an OSS/FS user interface is implemented using a web browser. This actually has a number of advantages: usually the user can use nearly any operating system or web browser, users don't need to spend time installing the application, and users will already be familiar with how their web browser works (simplifying training). However, web interfaces can be good or bad, so it's still necessary to evaluate the interface's useability.

If the application is a GNOME or KDE application, it is a good sign if the project is clearly trying to conform to its relevant user interface guidelines. For GNOME applications, make sure the application conforms to the [GNOME Human Interface Guidelines \(HIG\)](#). For KDE applications, make sure the applications conforms to the [KDE user interface guidelines](#). There have been some discussions on [merging the GNOME and KDE human interface guidelines](#) as part of the general work of [Freedesktop.org](#). Jim Gettys' [Open Source Desktop Technology Road Map](#) provides useful general guidelines on where open source desktops are going; it'd be useful to consider if the application is going with or against the current.

In the end, evaluating useability requires hands-on testing.

4.10 Security

Evaluating a product's security is complicated, in part because different uses and different environments often impose different security requirements on the same type of product. One step toward solving this problem is to [briefly identify your security requirements](#).

Some proprietary products undergo "Common Criteria" evaluation for security, however, the costs of this type of evaluation make it rare for OSS/FS products. If a Common Criteria evaluation *is* available, be sure to look at the product's "Security Target". The Security Target is publicly available document that specifies important information about the evaluation, such as exactly what configuration was tested, what assumptions were made, and what security requirements were tested. If the Security Target describes a configuration that's very different from what you will use, or doesn't include the security requirements that are important to you, the evaluation results may not be as applicable to you.

One hint is simply looking at the user's guide - does it discuss how to make and keep the program secure? Does the project have a process for reporting security vulnerabilities? Does the project have a cryptographic signatures (or at least MD5 hashes) for its current release? Another hint is to examine developer mailing lists to determine if they discuss security issues and work to keep the program secure. Examining databases of security vulnerabilities (such as MITRE's CVE) can help, but note that some of the most secure programs have historically had many vulnerabilities found due to extensive examination. Programs with no CVE entries may be relatively secure - or so insecure that no one's bothered to evaluate them.

Some projects are known for having a large number of reviewers specifically go through the code looking for potential flaws. [OpenBSD](#) in particular is famous for this; the [OpenBSD developers have spent a great deal of time performing a comprehensive file-by-file analysis of every critical software component](#). If the product is a critical part of OpenBSD, that's a good sign. However, the OpenBSD folks spend more time on critical infrastructure and server applications; not all applications in their packages (and especially their "ports") have gone through as thorough an examination, and not all changes OpenBSD makes are included in the original program used by others.

Some people look to see the nationality of the key developers, say by examining their email address and any biographies available about them, because they're concerned that some countries may intentionally insert malicious code. This has some value, but only up to a point. Clearly, an email address may be in one country but the person may actually be from another, and a person may be in one country yet be a citizen of another. But if you're concerned about this, do not use a different standard for OSS/FS than for proprietary development. Increasingly proprietary software development is done overseas by anonymous developers, and their code is not subject to public review. For example, [Microsoft has already conducted some research and product development in China, and in July 2002 committed to increasingly do so](#). Indeed, Microsoft's senior VP for the Windows division, Brian Valentine, specifically told Microsoft managers to [pick something from their project list "to move offshore today."](#) [In 2003, Oracle doubled the number of staff in India \(from 3,000 to 6,000\) for software development and other areas, and they noted that others are doing the same.](#) Even if development is done in one country, the developer may actually be a citizen of another. Even if a company is located in a country you trust, it is extremely unlikely it's performed a detailed, line-by-line review of code developed for it by citizens of other countries. As a result, an OSS/FS project headed by a foreigner may have far more development and review by locals than a proprietary program whose company headquarters happens to be located in your country. The concern is legitimate, but be sure that you have the same standard for all products.

Of course, if security is a major concern for you, then you will want to do more analysis of the top contending programs later. A later section of this paper [discusses evaluating software security in more detail](#).

4.11 Flexibility

Flexibility measures how well a program can be used to handle unusual circumstances that it wasn't originally designed for. In this attribute, OSS/FS programs have a significant advantage over most proprietary programs: any OSS/FS programs can be modified to handle any circumstance not previously considered before. However, taking advantage of this requires either programming skill, or convincing someone with that skill (usually by paying them) to modify the program. Also, some OSS/FS programs are easier to extend than others. You can at least look to see if there mechanisms that make the program easier to use for new purposes, such as "plug-ins", a programmer's application programming interface (API), or a command language. There are other ways that programs can be flexible, however.

4.12 Legal/license issues

Legal issues are another important attribute, and they are primarily defined by a program's license. Thus, you should examine the license requirements for each program you're considering, as well as their implications in your country.

Unlike most of the other attributes of software, this attribute is sometimes overlooked when evaluating proprietary software, and that's a mistake. When you're evaluating proprietary software, be sure to examine its licensing terms such as its End User License Agreement (EULA). Some EULAs have clauses that you may find unacceptable, such as allowing a vendor to gain access to your organization's computers and networks to do compliance audits, obligating you to large fines if the vendor finds unlicensed copies (even if the copies were not sanctioned by your organization), allowing the vendor to remotely disable your software without a court decision or other legal protection, forbidding the disclosure of evaluations (such as benchmarks) to others, limiting transfer or use of the program (such as limits on data volume), or allowing the proprietary program to send private information to the vendor. Even if others find the EULA conditions acceptable, you may not find the EULA conditions acceptable for your organization.

Of course, you should examine the license conditions for OSS/FS programs as well. OSS/FS programs' fundamental difference from proprietary software is the legal right of users to view, modify, and redistribute OSS/FS programs. However, many OSS/FS discussions revolve around software licenses. This is because different developers have different motivations for developing their software, and software licenses reflect their motivations. In particular, if you modify a program, some licenses require that you release the modifications under certain conditions; see the text on [copyleft](#) for more information.

Obviously, if an OSS/FS project might be shut down due to legal action, that would harm all of its users. Thus, it's worth checking to see if there are any pending legal actions against that project, and then consider its likelihood of success and impact should it succeed. Simply having some legal action is not necessarily an issue; widely popular OSS/FS projects sometimes attract frivolous lawsuits, just as widely popular proprietary vendors do. In that case, you'll need to examine the evidence (or at least examinations by technologists of the evidence) to determine if the issue is serious. Some actions may affect only certain countries, for example, software patents are often only relevant in certain countries. OSS/FS projects can usually move countries if they are forbidden by government action, but that may not help you if you're in the country where the use is forbidden. Obviously, if a legal action could completely shut down the project, and there appears to be a strong case for that legal action, that is an important risk.

In any case, since everyone's legal situation is different, if you have legal concerns you should consult a lawyer familiar with OSS/FS. Note that some intellectual property lawyers aren't yet familiar with OSS/FS, so be sure you get one who is knowledgeable about the issues before you pay for their advice. The text here is not specific legal advice, but it can at least help you identify some of the legal issues involved.

The paper "[A Comparison of the GPL and the Microsoft EULA](#)" by Con Zymaris compares the General Public License (GPL), the most popular OSS/FS license, with the EULA for Microsoft Windows XP Professional, a representative proprietary license. This was summarized in [a Sydney Morning Herald article](#).

This section discusses various license issues, emphasizing license issues specific to OSS/FS, and how to compare their terms to your needs.

4.12.1 Warranty/legal recourse

Some users are under the mistaken notion that they have significant legal recourse if their software doesn't work properly. Others are under the mistaken notion that proprietary software provides much greater legal protection than OSS/FS. This is almost never true; *both* OSS/FS and proprietary programs generally limit any legal liability so much that there isn't really a legal recourse in either case.

Most OSS/FS licenses specifically disclaim a warranty. The most common license, the GPL, disclaims warranty by default, but does mention that you may be able to purchase a warranty separately.

Proprietary programs are generally no different. Most disclaim any liability, or at best state that you may be able to get the purchase price back. Even obtaining a refund on proprietary software is difficult or impossible once it has been unwrapped, since vendors fear that users may have copied it first.

Con Zymaris examined the Microsoft XP Professional EULA (a sample proprietary license) and found that the license "explicitly removes all avenues and all recourse ... for legal relief of any sort. At best, you may recover the cost of the software product, or US \$5." It clearly states that users are not entitled to any damages, including consequential damages. In theory, countries may require that damages be paid anyway, but few countries' legal systems actually enforce such a requirement. Also, in theory the license permits refunds if the product doesn't work correctly in the first 90 days, but this doesn't work in reality for many proprietary programs. For example, obtaining refunds from Microsoft is extremely difficult, even if the software has never been used. [Many people have been unable to receive their promised refund on unused Microsoft products, despite a license that provides a refund and repeated court cases to compel Microsoft to honor the license.](#)

As a practical matter, few people can manage the legal resources to initiate a court case against a software vendor, and court cases against software vendors to obtain damages due to defective products almost never succeed. Therefore, it's unwise to depend on court cases to ensure that a given program will meet your needs.

4.12.2 License Audits

Users of proprietary software must set up organizations to track proprietary programs, and ensure that extra copies are not made. Otherwise, they risk being sued for piracy. Since it's usually quite easy to copy programs, this is not an easy task to do. In some cases, users of proprietary programs must bear the risk and costs of later license audits. Increasingly, to ensure that customers do not have unauthorized copies, vendors are relying on license audits to ensure compliance.

4.12.3 License issues unique to OSS/FS

However, there are some license issues unique to OSS/FS; this section will discuss them in more detail.

4.12.3.1 Checking if the program is OSS/FS

A first step is to quickly determine what the license(s) of the programs are, and then check if they are truly OSS/FS licenses. Obviously, if the license isn't an OSS/FS license, then much of the material in this paper won't apply. The most common OSS/FS licenses include the [General Public License \(GPL\)](#), the "[Library](#)" or "[Lessor](#)" [General Public License \(LGPL\)](#), the "BSD-style" license, and the "MIT-style" license. Unfortunately, recognizing the latter two licenses is a little harder, because the text of these licenses changes in every case (in particular, the names of programs and organizations are inserted). To recognize the BSD-style and MIT-style licenses, compare their license texts to the standard boilerplate text of the [BSD](#) and [MIT](#) licenses.

If it's not one of those extremely common licenses, consult the [Open Source Initiative \(OSI\)'s list of open source software licenses](#) and the [Free Software Foundation \(FSF\)'s list of Free Software licenses](#). If it's OSS/FS, then the license is very likely to be on at least one of those lists.

If that doesn't work, look at the license - it may quickly become obvious that it's not OSS/FS at all. The license may prevent you from freely redistributing original or modified versions of the program, it may charge a per-use fee, or its use may be restricted to "non-commercial use only" (OSS/FS programs must be useable for commercial purposes without restrictions). Microsoft's "Shared Source" licenses, for example, are not OSS/FS licenses.

Failing that, you may need a lawyer familiar with OSS/FS issues to truly determine if a program's license is OSS/FS. In practice this last step is almost never needed; nearly all OSS/FS programs are distributed under a very small set of common licenses.

4.12.3.2 Why different OSS/FS licenses matter

Licensing issues are important to developers, but for the majority of users in many circumstances they don't actually matter. OSS/FS licenses are far more different from typical proprietary licenses than from each other. All OSS/FS licenses permit users to use the software, modify it, and redistribute the original or modified version as much as they like. Since most users don't modify their software, and since the differences between OSS/FS licenses primarily affect developers, users don't notice the differences in most cases.

However, licenses *are* an important issue to consider for OSS/FS, because there are a few cases where they definitely *do* matter. OSS/FS licenses don't impact users who simply use the software as-is; users who do not change the program can freely copy it to anyone

else they wish. However, licenses do impact those who modify the program, and the license can even impact users who don't modify their programs in the sense that the licenses impact what they (and those they work with) can do later.

4.12.3.3 Copylefting vs. non-copylefting

There are fundamentally two kinds of OSS/FS licenses: "copylefting" licenses and "non-copylefting" licenses. A program released under a copylefting license allows anyone to change the program - but those changes must be provided to recipients under exactly the same conditions as the original. In other words, an OSS/FS program released under a copylefting license cannot be later turned into a proprietary program by a third party. Most OSS/FS software is released under copylefting licenses, such as the General Public License (GPL) and the Lessor/Library General Public License (LGPL).

There is a long argument of the advantages of copylefting vs. non-copylefting licenses. Some view non-copylefting licenses as "more free" because recipients of that code can do anything they want with the code - including making a modification and producing a proprietary version. However, many others view copylefting licenses as "freer" than non-copylefting ones, because they ensure that all later recipients of modified versions can also modify and maintain the code. Another way of looking at this is that copylefting licenses ensure that all later users and developers have more freedoms, at a cost of giving fewer freedoms to the immediate recipients.

Non-copylefting licenses allow proprietary vendors to easily incorporate the code into their products, and modify it any way they like. If the goal of the OSS/FS project is to promote widespread adoption of a new protocol or data format, including in proprietary products, then the license clearly promotes the goal. Thus, non-copylefting licenses are often used when the goal is to promote the adoption of a standard. An example of this approach is the implementation of the Internet's suite of standards (often called the "TCP/IP standards"); the key code was licensed under a BSD-style license, and it has certainly become ubiquitous. Even developers who re-implement a standard can find it helpful to have code, because implementation code can clarify issues that are usually left ambiguous in a standards' text.

However, an OSS/FS program that is not copylefted is often modified and incorporated into a proprietary product. Often, the proprietary vendor will never release any code improvements back to the OSS/FS project. If this action is repeated (and there are economic incentives to doing so), the OSS/FS project will be significantly weakened. Over time, the proprietary vendor may add other functionality to the program, or even intentionally make incompatible changes to prevent users from using the original program. Such strategies - especially the latter one - are called "embrace, enhance, and extinguish". Indeed, the proprietary extension might displace the original OSS/FS project, at least for a while, if the OSS/FS program does not use a copylefting license. As a result, projects without a copylefting license can sometimes have longevity problems or functionality weaknesses, because there are economic incentives for commercial vendors to compete with the OSS/FS project instead of cooperating with it. Many have argued

that this has been a problem with the *BSD operating systems: the *BSDs were ready for use before the Linux kernel was, but proprietary versions (such as SunOS and BSDI) often did not release their proprietary extensions back to the OSS/FS project. In contrast, the Linux kernel's copylefting GPL license encourages competing vendors to cooperate because there's no economic advantage to dividing. Thus, the Linux kernel added far more capabilities (such as symmetric multiprocessing and more device drivers) because there was no advantage to doing otherwise.

This issue of which license is "better" is a long-standing debate, and not one that will be settled in this paper. Indeed, sometimes the same developer will choose different licenses for different products, depending on that developer's motivations. For example, I use both copylefting and non-copylefting licenses when I release OSS/FS code, depending on my purposes. In short, differences in the OSS/FS license reflect differences in motivation of the original developer, and will impact how the program can be used and supported.

4.12.3.4 Computer Libraries

The biggest issue with copylefting licenses involves computer libraries. A little background information is needed to understand how they affect things. First, all computer programs build on computer libraries, which may build on other libraries; some computer libraries act like the foundation of a house, and anything that affects a foundation can affect everything else. Any developer of a program must ensure that all the libraries they use have compatible licenses. This is actually no different than with proprietary programs - proprietary program developers have had to check their library licenses for compatibility for many decades (e.g., to identify royalty payment requirements).

In particular, if a computer library is covered by the GPL (unamended), then any program that uses that library must also be released under the GPL. Thus, if you are intending to build a proprietary system and wish to use a library, you generally cannot use a library under the GPL. For example, the [readline](#) library has some nice routines for letting people modify typed-in commands, and the [GNU Scientific Library \(GSL\)](#) has useful scientific routines, but both are released under the GPL. This isn't a problem if you're willing to release your resulting program under the GPL, but this is a problem if you are not willing. This is not a "bug"; this is exactly what the developers of the GPL intended, because they want to encourage other developers to use the GPL. You *can* build proprietary programs using a library released under the LGPL; the LGPL allows use by proprietary programs, but any changes to the library itself must be released under the LGPL. In contrast, proprietary programs can often use GPL'ed programs (not libraries), though the details are beyond the scope of this paper. In short, if you intend to use any library (proprietary or OSS/FS), you must check the library's license before incorporating it into your program.

Many OSS/FS licenses are compatible with others, but this is by no means true for all licenses. [The most popular OSS/FS license is the GPL, and I argue in another paper that all developers of OSS/FS programs should strive to use a GPL-compatible license.](#) For

example, it is generally accepted that it's fine to combine code licensed under the GPL, LGPL, MIT, and BSD-new licenses (four of the most common OSS/FS licenses); the combination as a whole would have to be released under the GPL license.

However, the older version of the BSD license (sometimes called BSD-old) is incompatible with the GPL according to the GPL's developers. The older BSD license included a clause called by some the "obnoxious BSD advertising clause": "All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed by the University of California, Berkeley and its contributors." As documented by the FSF, [the real problem was that everyone else added their own name to the acknowledgement list, creating practical problems as programs got large](#). In June 1999 the University of California removed this clause from the license of BSD, and at this point most programs using the older BSD license have switched away from it. However, there are a few programs which still use the older license, and this can be a source of problems.

4.12.3.5 Other examples of license impacts

Sometimes the computer library issues can indirectly affect users, even though the users aren't writing any code themselves.

One of the few cases where this hits users unexpectedly is in choosing a graphical desktop environment. There are two competing OSS/FS graphical desktop environments: GNOME and KDE. GNOME is based on the GTK+ library (released under the LGPL), while KDE is based on the Qt library (released under GPL license, and also available with a proprietary license for a fee). This means that users who choose use GNOME can develop proprietary programs (or have them developed) using the "native" library without paying an additional fee, while users who choose KDE and use proprietary programs using the "native" library will pay additional fees. Why? That's because developers of proprietary KDE programs must pay additional royalty fees to the company who can grant proprietary licenses for Qt.

Whether or not this is an issue depends on your point of view. Proponents of this approach (some KDE developers) say this enables the Qt developers to have a viable business model. Indeed, to their credit, the Qt developers did have to even release their product for Linux and Unix under a GPL license, and they decided to do so. Detractors state that this arrangement allows a single company to monopolize control over the fundamental infrastructure of the KDE desktop. For example, it would be difficult for anyone else to change Qt and have it widely used, since only the original Qt developers can release the version used by proprietary products. Note that GNOME and KDE can actually run the applications of the "other" environment (as long as the necessary libraries are installed), though sometimes not quite as cleanly. Also, this distinction is entirely irrelevant to programs released under the GPL. Thus, this distinction is not as clear-cut as it first seems.

In most cases, however, these licensing differences don't matter to ordinary users. For example, users can easily use the Linux kernel, which is released under the GPL, to run proprietary programs. Proprietary programs and OSS/FS programs can be used on the same computer without difficulty. This doesn't mean that users should ignore the issues, but the differences between the various OSS/FS licenses are far smaller than the differences between typical proprietary licenses and OSS/FS licenses.

4.12.3.6 Patent defense

Another relevant issue is patent defense. Some OSS/FS licenses require that, if you modify the program in a way that it implements a patent, and you own the patent, then you must grant all recipients the rights to use the patent. Obviously, if you own any patents, you should ensure that you wish to give this grant before you make changes to programs covered by patent defenses.

4.12.3.7 License summary

In summary, OSS/FS software licenses are important to developers, and they can impact users who may become developers (or pay developers to make a change). However, OSS/FS software licenses primarily cover what developers can and cannot do, not of users who do not change the software.

4.13 Other Issues

Of course, there may be other issues that are important to you, such as local policies or unusual technical requirements. Clearly you should include those in your evaluation as well.

Some organizations have specific policies that state they are neutral about OSS/FS. [The U.S. Department of Defense specifically states that OSS/FS may be used as well as proprietary software, all under the same rules.](#)

In contrast, some governments and other organizations have specific policies preferring OSS/FS programs over proprietary programs. At the time of this writing, this is rare; what's unclear is whether or not this is a trend. [The Economist identifies reasons some governments choose to prefer, mandate, and/or fund OSS/FS programs:](#) some governments are reluctant to store official records in the proprietary formats of proprietary software vendors, they believe the software's transparency increases security because security problems can be quickly exposed and fixed, the software can also be tailored to the user's specific needs, upgrades happen at a pace chosen by the user (not the vendor), and this move tends to benefit numerous small, local technology firms.

Even without a specific policy, some organizations may particularly like the control that OSS/FS programs give them (e.g., because they can modify the program to suit their needs or to remove a vulnerability immediately). You should identify which programs are

OSS/FS, regardless, but identifying those programs would be especially important in such cases.

You should make sure, however, that these "other issues" are really issues. For example, some organizations strongly prefer "commercial" software. Some managers fail to realize that OSS/FS programs are nearly always commercial software as the term is often defined: the software is made available to the public, with support available to the public. The OSS/FS programs you are most likely to evaluate have at least one company available which sells support contracts of various kinds. Most organizations preferring "commercial" software are simply trying to avoid programs that are entirely created and supported internally, since such approaches are very expensive. OSS/FS projects avoid these expensive approaches as well, just in a different way than proprietary programs avoid them.

Another problem is confusing OSS/FS programs with "shareware" or "freeware" programs, since some organizations have policies against shareware or freeware. OSS/FS programs are not "shareware" or "freeware" as the terms are usually used; these terms are usually used to describe distribution mechanisms for proprietary software. For example, typically the source code for shareware and freeware programs is not available, and these programs do not permit later modification by others. This has many practical ramifications: support is often difficult to acquire (especially for freeware), support cannot be practically competed, security problems cannot be easily identified and addressed, end users cannot modify the program to fix errors or add functionality, and so on. Some may refer to OSS/FS programs as "freeware", but mixing terminology like this is more confusing than helpful.

5. Perform an in-depth analysis of the top candidates

After this initial evaluation, you then pick the top contenders, and perform a more in-depth analysis of them. In particular, get them and try them out on representative work loads.

This step is, for the most part, done the same way for both proprietary and OSS/FS programs. The important attributes to consider are the same as in the previous step; you simply spend more effort by actually trying things out instead of quickly reading the available literature. For example, to see what functionality a program provides, you'd run it and try out the functionality that you're interested in using (e.g., if you're concerned about interoperability, acquire some sample same files or systems and see how well it works).

For performance and scalability, set up a representative situation dummy data (and a dummy amount of data) and see how well the program performs. For many circumstances, today's higher computer speeds mean that performance is often not as

important as it used to be, but there are still applications where performance matters. If performance is important, try to make your test circumstance as realistic as practical, because the same program can perform well in another's situation yet perform poorly in yours. For an example, see [Bradley J. Bartram's article on stress testing Apache-based web applications](#). [Opensourcetesting.org](#) maintains a list of [OSS/FS performance test tools](#) which may help.

Obviously, you should try any new program in non-critical situations first, to see how well it works before truly deploying it.

If you are dealing with large amounts of money, there are some additional issues to ensure you get the best price. For proprietary or OSS/FS programs you should investigate getting an enterprise license, or negotiating a special rate from suppliers. If the number of units is large, you may want to do some support in-house (using your existing support organization to answer "easy" common questions) and then only pass on new (hard) questions to a support vendor; that may save a large amount of money. Negotiate with what you perceive as your top options, but do *not* make a decision until after all offerers have given their best deal - and make it clear to each that you have other options. For example, it's widely known that [Microsoft offers significant discounts to some organizations to avoid losing bids to GNU/Linux](#). You can overpay by millions of dollars if you fail to diligently use competition to get your best deal.

You should always carefully identify the version number of the program, because what you say about one version may not be true in a later version. This is particularly important for OSS/FS programs, because many OSS/FS programs undergo rapid improvement.

There are a few differences for OSS/FS at this step, however. For example, getting an OSS/FS program is sometimes faster, since you can often simply download the full version. In contrast, some proprietary programs require shipping since they are only available in shrink-wrapped packaging.

A more important difference is that there are sources of information about an OSS/FS program that may not be available for proprietary software. In particular, you can also have a software professional examine the program's design documentation, source code, and other related materials. You can do this for any reason, but this paper will examine two particularly common cases: considering the possibility of adding functionality, and examining the program's security.

5.1 In-depth Analysis for Adding Functionality

If the OSS/FS program had some but all the functions you need, you should examine what it would take to add those functions. This can be done by paying others, or by doing it in-house.

If you are considering doing it in-house, have a software professional examine the program's design documentation and source code to see how well it's put together. Well-designed programs are easier to understand and modify. Make sure they know what functions you're interested in adding; that will enable them to see how much effort there would be in adding those functions. Make sure you talk with the project developers; there may be an ongoing project to add some of those functions.

5.2 In-depth Analysis of Software Security

Another area where examining the software code can be particularly valuable is when you want to carefully evaluate a program's security. You can have software development experts look at the code to see if the OSS/FS appears to be trustworthy (e.g., if it follows good practices). For example, they could see if:

1. it minimizes privileges (e.g., only small portions of the program have special privilege, or the program only has special privileges at certain times)
2. it strives for simplicity (simpler designs are often more secure)
3. it carefully checks inputs
4. source code scanning tools such as RATS and [Flawfinder](#) report few problems.

Of course, your software development experts will need to know how to develop secure software in the first place. Unfortunately, this information is usually not taught by schools or development organizations. If they don't already know how to do this in detail, make sure they learn how to do so first. [My book on how to write secure programs](#) is freely available and has a lot of detail on the topic.

Another approach is to hire a commercial lab to perform a Common Criteria evaluation of the product. At this time users usually don't pay for a Common Criteria evaluation of an OSS/FS product (due to the time and expense), but it's certainly possible.

6. Wrap-up

Both OSS/FS and proprietary programs can be evaluated, using essentially the same approach. However, the way you acquire the information to evaluate OSS/FS programs is often different, because OSS/FS projects tend to produce different kinds of information that you can use for your evaluation. There's no guarantee that following this process (or any other process!) will always find the "best answer", but I believe you have a good chance of getting a reasonable answer by following this process.

If your management wants it, you should be able to quickly present the results of your evaluation (say, as a few slides in a short presentation). Here's one possible outline of such a presentation:

1. The problem (what were you looking for?)

2. The recommended solution, along with a brief statement as to why you believe it's the right answer. Some people wait until the end to give "the answer", but I believe it's much better to state the final answer immediately. That way, management can follow the rest of the presentation to see how it justifies that answer; without this focus, your audience may lose interest or get lost in the details. Most people don't want to wait to get this information; in fact, if there's no controversy, you may be able to save time for everyone by stopping at this point!
3. The process used to get to this recommendation. In particular, identify the four basic steps as given here (identify, review, compare, analyze) and how long you spent doing the evaluation. Please credit this paper as your process, if you use it, as *How to Evaluate Open Source Software / Free Software (OSS/FS) Programs* by David A. Wheeler, http://www.dwheeler.com/oss_fs_eval.html.
4. Identify the main alternatives (including the recommended alternative), with a brief description of each alternative. Be sure to identify the version number of each program; some OSS/FS programs undergo rapid improvement, so what you say about one version may not be true in a later version. You should rank the alternatives from "best" to "worst" if you can, so that the managers can focus on the most probable alternatives. Some people assign different weights to attributes, measure each product how well they meet those attributes, and then roll those values up into a final quantitative score. For example, you might have weights of 1 to 5 (5 is most important), have scores from 1 to 5 (5 is completely meets the requirement), multiple each score by its weight to give a weighted score, and each product could be rated as the sum of its weighted scores (where a larger score is better). Even if you don't have a formal scoring system, you will need to know what attributes are most important for your circumstance. In some cases you may need to list the same program more than once, e.g., to compare using an OSS/FS program as-is versus modifying the program to add an especially desirable function.
5. If desired, discuss each alternative in turn in more detail. Different managers will want different levels of detail, and clearly the level of detail will depend on the importance of the decision.
6. Conclude again with the final recommendation, with a brief statement as to why you believe it's the right answer.

If the most promising OSS/FS project is moribund, you can restart or offer to re-lead the project; however, this is a much more significant commitment. If no OSS/FS project seems to meet your needs at all, you can certainly consider developing an OSS/FS program to meet the need - again, this requires a more significant commitment. You will still need to reuse libraries to get the job done, so you can use the process in this paper to evaluate those libraries.

Once a decision has been made, it's time to begin the process to install the new program. A good paper on how to migrate from proprietary programs to OSS/FS programs is Interchange of Data Administrations (IDA)'s paper, [The IDA Open Source Migration Guidelines](#).

There are now many areas where a useful OSS/FS program is already available. Hopefully, this document will help you evaluate your options.

For more information, see [Why Open Source Software / Free Software \(OSS/FS\)? Look at the Numbers!](#), [Generally Recognized as Mature \(GRAM\) OSS/FS Programs](#), [Open Source Software / Free Software \(OSS/FS\) References](#), and [David A. Wheeler's home page](#).

About the Author

David A. Wheeler is an expert in computer security and has a long history of working with large and high-risk software systems. His books include [Software Inspection: An Industry Best Practice](#) (published by IEEE CS Press), *Ada 95: The Lovelace Tutorial* (published by Springer-Verlag), and the [Secure Programming for Linux and Unix HOWTO](#). Articles he's written include [Why OSS/FS? Look at the Numbers!](#), [More than a Gigabuck: Estimating GNU/Linux's Size](#) and [The Most Important Software Innovations](#). Mr. Wheeler's web site is at <http://www.dwheeler.com>; you may contact him at dwheeler@dwheeler.com, but you may not send him spam (he reserves the right to charge fees to those who send him spam).

