INSTITUTE FOR DEFENSE ANALYSES

Using an Open Source Software Approach for Cybersecurity Technology Transition

David A. Wheeler

November 24, 2015

Approved for public release;
distribution is unlimited.

IDA Paper
P-5279

Log: H 15-000816
Copy

INSTITUTE FOR DEFENSE ANALYSES
4850 Mark Center Drive
Alexandria, Virginia 22311-1882

The Institute for Defense Analyses is a non-profit corporation that operates three federally funded research and development centers to provide objective analyses of national security issues, particularly those requiring scientific and technical expertise, and conduct related research on other national challenges.

About This Publication

This work was conducted by the Institute for Defense Analyses (IDA) under contract N66001-11-C-0001, subcontract D6384-S5, Project GT-5-3329, "Homeland Open Security Technology (HOST)," for the Georgia Tech Research Institute. Funding for the report was provided by the Department of Homeland Security. The views, opinions, and findings should not be construed as representing the official position of either the Department of Defense or the sponsoring organization.

Acknowledgments

Samir Khakimov, Margaret E. Myers, Clyde G. Roby

INSTITUTE FOR DEFENSE ANALYSES

IDA Paper P-5279

Using an Open Source Software Approach for Cybersecurity Technology Transition

David A. Wheeler

Executive Summary

Open source software (OSS) is "software for which the human-readable source code is available for use, study, reuse, modification, enhancement, and redistribution by the users of that software." [DoD2009] An OSS approach is an approach to software research and development (R&D) that releases OSS and encourages the collaborative development of OSS. Using an OSS approach can enable and speed technology transition, but many researchers and government program managers are unfamiliar with how to use an OSS approach to support technology transition.

This document provides guidance on how to use an OSS approach to support technology transition. The goal of this guide is to help researchers (including principal investigators) turn their ideas into successfully deployed solutions, and also to help program managers as they select research proposals and oversee researchers' work. This guide focuses on aiding technology transition for work supported by the Department of Homeland Security (DHS) Science and Technology (S&T) Directorate, particularly its Cyber Security Division (CSD), which focuses on defensive cybersecurity research. This includes research on developing trustworthy cyber infrastructure, on foundational elements of cyber systems, on cybersecurity user protection and education, on research infrastructure to support cybersecurity, and on cyber technology evaluation and transition (per http://www.dhs.gov/csd-program-areas). However, portions of this guide may also be useful to others performing technology transition of government-funded research.

There are many potential advantages for using an OSS approach. The Department of Defense (DoD) has identified many positive aspects of OSS, particularly for its users. In particular, "the continuous and broad peer-review enabled by publicly available source code supports software reliability and security efforts through the identification and elimination of defects that might otherwise go unrecognized by a more limited core development team. The unrestricted ability to modify software source code enables [users] to respond more rapidly to changing situations, missions, and future threats…OSS is particularly suitable for rapid prototyping and experimentation, where the ability to 'test drive' the software with minimal costs and administrative delays can be important." [DoD OSS 2009]. Many quantitative studies show that OSS programs are often a reasonable or superior alternative to their proprietary competition according to measures such as market share, reliability, performance, scalability, security, and total cost of ownership [Wheeler2014a].

OSS is often mistakenly called "non-commercial." OSS almost always meets the definition of a "commercial item," as that term is defined under U.S. procurement law (41 U.S.C. § 103(1)). Under this definition, an item qualifies as a "commercial item" if it is not real estate, is customarily used for non-government purposes, and is either licensed to the public or offered for license to the public. Since nearly all OSS meets these criteria, nearly all OSS can be considered commercial computer software.

This guide begins with an introduction to the basics of technology transition, open source software, and applying OSS approaches to technology transition. It then discusses how to build on existing work, including how to evaluate existing software and modify existing OSS. This guide then describes how to establish a collaborative environment for a new OSS project when necessary, including selection of hosting services, a governance process, and license, and dealing with the issues of contributor agreements and contributor assignments. The key issues in OSS project inputs and results are then discussed, including processes for accepting technical contributions and a variety of common conventions. Tips for making an OSS project successful are then provided, focusing on those that have been statistically verified as really working. The document closes with approaches for overcoming common impediments.

Simply developing and releasing software can be useful (e.g., for documenting past research so it can be repeated), but it does not necessarily create a successful OSS project. The guidelines in this document should increase the probability of success. However, there are some potential impediments to OSS projects that are common and have specific corrections. These include the following:

Users/developers do not know of the project

A common problem is that users and potential co-developers are unaware of the OSS project. A clear (preferably unique) project name can help them find the project, once they know of it, but a unique name will not (by itself) help them discover that it exists in the first place. It is critical to have a front-page website that clearly describes the purpose of the project; this can help those who are searching for it. This will help those specifically seeking projects like it, especially once others start linking to the project front page (since this increases search rankings). However, many users will not know to seek out the project, and obscure projects often have low rankings in web searches.

Missing functionality

All software could have additional functionality added, and many users will want functionality that the current software lacks. There are steps that can make missing functionality less of an impediment in some cases; they can be grouped into those making it easy to work around missing functionality and those making it easy to add that functionality.

Inadequate quality or trustworthiness

Users are far less likely to use buggy software, especially if those defects directly affect their primary reasons for using the software. OSS projects should use a variety of techniques to provide and maintain good quality. This includes general quality issues (e.g., functions that perform incorrectly), but also security issues (e.g., enabling attack). This is even true for security-related programs; a security-related program can itself have vulnerabilities that enable instead of counter attack. Since no single mechanism guarantees high quality, a variety of approaches should be used; some of them have already been noted above.

Low trust

Users may not trust a project, even if the code itself is trustworthy. Thus, it is important to make it clear to users why they should trust the project. People are often more willing to trust a project if many others trust it. Thus, focusing on improving the software so that it will help many users – and growing that user base – can also help gain the trust of others.

Excessive user cost/time/effort

Potential users have limited time, and co-developers in OSS projects often start as users. Thus, any barrier for user actions can lead to dramatically fewer users and developers. Consider tasks from the user's point of view and constantly work to reduce their cost, time, and effort. Efforts to reduce cost, time, and effort must typically continue over time. The good news is that once a project begins gaining contributions (because it has reduced that effort to some reasonable level), other contributions can help reduce the effort further.

An OSS approach can be a very effective way to perform technology transition of cybersecurity R&D. However, as with any approach, it needs to be applied well; this guide should help researchers and PMs successfully apply it.

Contents

Figures and Tables

1. Introduction

The United States and its allies depend on computer and network systems, and these are currently under attack. To address this challenge, the U.S. Government actively invests in cybersecurity research. However, a serious problem with today's cybersecurity research is inadequate technology transition. Technology transition is the set of all efforts to ensure that technologies developed in research settings will eventually be deployed and used operationally [Maughan2013]. A newer approach to transitioning technology based on open source software (OSS) is now available. OSS approaches have the potential to improve technology transition in many cases. However, OSS approaches only work well when they are properly applied.

OSS can be defined as "software for which the human-readable source code is available for use, study, reuse, modification, enhancement, and redistribution by the users of that software." [DoD2009] An OSS approach is an approach to software research and development (R&D) that releases OSS and encourages the collaborative development of OSS. As explained further below, using an OSS approach can enable and speed technology transition. However, many researchers and government program managers are unfamiliar with how to use an OSS approach to support technology transition.

This paper provides guidance on how to use an OSS approach to support technology transition. The goal of this guide is to help researchers (including principal investigators) turn their ideas into successfully deployed solutions, and also to help program managers as they select research proposals and oversee researchers' work. This guide focuses on aiding technology transition for work supported by the Department of Homeland Security (DHS) Science and Technology (S&T) Directorate, particularly its Cyber Security Division (CSD), which focuses on defensive cybersecurity research. This includes research on developing trustworthy cyber infrastructure, on foundational elements of cyber systems, on cybersecurity user protection and education, on research infrastructure to support cybersecurity, and on cyber technology evaluation and transition (per http://www.dhs.gov/csd-program-areas). However, portions of this guide may also be useful to others performing technology transition of government-funded research.

This guide focuses on two key roles:

- Researcher: A researcher proposes research and development (R&D) to be performed, performs the R&D if selected, and/or supports individuals performing R&D. This role includes the principal investigator, any other investigators, software developers, and all other individuals in their organization

who support them. It also includes the organizations these individuals work for or represent, including their subcontractors at all tiers. The offeror is a researcher (or the organization the researcher works for) who develops a proposal for evaluation by DHS S&T. There is no specific limit on who a researcher works for, if anyone. Thus, a researcher may work for a university, a government (federal, state, local, tribal, or non-U.S.), a non-profit company, or a for-profit company. The specific individuals and their involvement may change over time, e.g., as a technology matures, a researcher whose primary skill is leading the development of production software may become more involved or become a leader of the effort.

- Program manager (PM): A PM selects research proposals to be funded and/or oversees the selected work as it proceeds. The PM also determines whether additional phases of the work should be funded. Generally, PMs are government employees or their delegates.

This guide first briefly describes technology transition and OSS concepts in general. It then provides a brief overview of how to apply an OSS approach to technology transition. It then focuses on the basics of applying OSS approaches to technology transition in general. It then discusses how to build on existing work, including how to evaluate existing software and modify existing OSS. This guide then describes how to establish a collaborative environment for a new OSS project when necessary, including selection of hosting services, a governance process, license, and the issues around contributor agreements and contributor assignments. The key issues in OSS project inputs and results are then discussed, including processes for accepting technical contributions and a variety of common conventions. Tips for making an OSS project successful are provided, focusing on those that have been statistically verified as really working. The document closes with approaches for overcoming common impediments.

Researchers and PMs who plan to use an OSS approach should incorporate their OSS plans as part of their proposal, since planning ahead can reduce the risk, cost, and time for technology transition. That said, an OSS approach can be applied even after a proposal has been accepted without such plans, although some actions may be more difficult.

This guide focuses on technology transition issues specific to applying an OSS approach. Other papers, such as [D'Amico2013], provide general guidance for technology transition regardless of whether or not an OSS approach is being used. Many papers and books provide tips for establishing and running an OSS project including [Scott2011], [Fogel2009], [Raymond2000], and [Gabriel2005]. We include some of these tips also noted elsewhere because they are important. However, this document emphasizes recommendations that either are especially important to researchers or are recent conventions or lessons learned that are not always mentioned in older guides.

Specific technologies and organizations are mentioned in this guide, but no endorsement is implied.

2. Basics of Technology Transition

As noted earlier, technology transition is the set of all efforts to ensure that technologies developed in research settings will eventually be deployed and used operationally [Maughan2013]. This section briefly explains why technology transition is important, then it discusses how an OSS approach can be used to help perform technology transition. This section concludes with a description of the *Need, Approach, Benefit, and Competition* (NABC) framework; this is a simple way to quickly communicate an idea and its value proposition.

A. Technology transition is important

The cybersecurity problem is large and growing; both government and industry are routinely victims of severe attacks. New and innovative solutions are urgently needed; however, merely creating new solutions is useless. "New and innovative technologies will only make a difference if they are deployed and used. It does not matter how visionary a technology is unless it meets the needs and requirements of customers and users, and it is available as a product via channels that are acceptable to the customers and users…We cannot afford to have technologies be put on a shelf because the funded projects ended and the researchers moved on to new problems that were yet unsolved." [Maughan2013].

Technology transition is the set of all efforts to ensure that technologies developed in research settings will eventually be deployed and used operationally [Maughan2013][1]. DHS S&T focuses not just on creating technology, but also on successfully transitioning technology to users. Clearly it is important to the government, and the public the government serves, that technology be successfully transitioned so that government investments can help solve cybersecurity problems. In addition, many researchers and PMs want more than the publication of academic papers; they want their ideas to be put into practice so that their work can actually help people.

Technology transition does not just happen. Everett M. Rogers, in his book *Diffusion of Innovation,* states that "technology transfer is difficult, in part, because we have underestimated just how much effort is required for such transfer to occur

[1] In this paper we use the term technology transition broadly (just as [Maughan2013] does). We do not make the distinction between technology transition and technology transfer that is sometimes used in the DoD community.

effectively" and that "technology transfer is usually a two-way, back-and-forth process of communication" [Rogers2003,pp150-152]. The gulf between creating technologies and implementing them is often so large, and fails so often, that it is sometimes referred to as the "valley of death." The "introduction and acceptance of new technology often depend more on social, cultural, and historical factors than on technological merit…once technologies become entrenched, change is very difficult to effect…volumes have been written about failures in technology transition and the disastrous consequences that befall [organizations] that fail to recognize and adopt pivotal new technologies" [NRC2004].

DHS S&T enables technology transition through pervasive emphasis, early involvement, active engagement, and tangible support. There is no single path to successful technology transition, so DHS S&T uses a variety of approaches, including establishing a government-funded project, establishing a proprietary product or service, and establishing an OSS approach for technology transition.

B. An open source software approach can be used to help perform technology transition

An OSS approach can be used to help perform technology transition. "For all R&D [research and development] sources of new technology, open source is an alternative to traditional transition channels. A number of government programs encourage or require technology to be released under open source licensing, as part of the R&D activities. Open source availability is well documented as a powerful and effective means to bring important capabilities into adoption, use, and support by larger communities" [Maughan2013].

An OSS approach provides a mechanism to quickly make results available to many potential users, including other researchers (who may be able to build on that work and innovate further) and end users (who need implementations of these research ideas). In all cases an OSS approach enables the two-way, back-and-forth process of communication that [Rogers2003] identifies as being critical for technology transfer. This quick availability of results and two-way communication is of benefit to PMs (who want to see the work they fund made available to users) and to researchers (who along with seeing their work being employed, will benefit from association with a well-known OSS project, which can increase the likelihood of future funding through growth in reputation).

An OSS approach can also counter some of the problems that inhibit technology transition:

- Some new technologies can be difficult to apply because potential users do not understand what they do and why they work. Papers can help in some cases, but all too often papers fail to reveal key issues relevant to potential users. OSS

makes it easy for potential users to examine exactly what a technology does and how it works.

- Some technologies apply to only a limited set of circumstances. OSS makes it easier to determine those limitations, and also makes it possible to make changes to the software that may expand its application.
- Some implementations of a technology are not rigorously tested. OSS makes obvious what its regression test suite covers and enables potential users to avoid or fix the untested areas as appropriate.
- Some implementations must be changed to meet the needs of users. For example, changes may be required to meet changing requirements, to work under different than intended circumstances, to add functionality, to make it more efficient, or to interface with other systems or components. OSS enables users to make those changes (directly or by contracting others to do so).

In some solicitations, DHS S&T has declared "a strong preference for open source licensing of software for all software developed and delivered and the licenses for all proposed software deliverables will have to be identified in submitted white papers and proposals," although "researchers may also offer a strong technical transition plan for deployment of the technologies developed" as an alternative to OSS release [DHS-Mobile2014] [Walker2014].

All technology transition approaches (including an OSS approach) need to be considered early, while the technology is still in development. Otherwise, there is a risk that the transition will be delayed by years, if it occurs at all. Even if a technology appears to be non-viable, transitioning it to OSS enables others to review it to either replicate the results or find an improvement that might make it viable.

Note that using an OSS approach for technology transition is an example of open security. Open security is "the application of open source software (OSS) approaches to help solve cyber security problems. An OSS approach collaboratively develops and maintains intellectual works (including software and documentation) by enabling users to use them for any purpose, as well as study, create, change, and redistribute them (in whole or in part). Cyber security problems are a lack of security (confidentiality, integrity, and/or availability), or potential lack of security (a vulnerability), in computer systems and/or the networks they are a part of." [Wheeler2013a]

C. Need, Approach, Benefit, Competition

The NABC framework is a simple way to quickly communicate an idea and its value proposition. NABC was promulgated in a book, *Innovation – The Five Disciplines for Creating What Customers Want*, by Carlson and Wilmot [Carlson2006]. SRI

International uses NABC to identify and refine an idea with potential business merit [SRI2011]. In the article "Developing Better Value Propositions Using the NABC Framework," [Schindlholzer2008] describes the NABC framework as one that answers the following questions:

1. Need: What is the most important customer and market *Need*?
2. Approach: What is the unique *Approach* for addressing this need?
3. Benefit: What are the specific *Benefits* per costs that result from this approach?
4. Competition: How are the benefits per costs superior to the *Competition's* and other alternatives?

In this document, we will use NABC to examine applying OSS to an entire project, and also use NABC to examine options that support a given project. In many cases, we will not specifically use the NABC terms (need, approach, benefit, and competition) but simply imply them through prose to make the text easier to read.

3. Basics of Open Source Software

This section briefly describes the basics of OSS for those less familiar with it. This section describes what OSS is, how it is typically developed, examples of OSS projects, and some reasons for using an OSS approach; counters some misconceptions about OSS; and briefly notes legal requirements for releasing software as OSS when it is developed with government funding. This is only a brief introduction; see the citations for more information.

A. What is open source software (OSS)?

As noted earlier, Open Source Software (OSS) is "software for which the human-readable source code is available for use, study, reuse, modification, enhancement, and redistribution by the users of that software." [DoD2009] The Open Source Initiative (OSI) (http://www.opensource.org) publishes the widely used "open source definition," a lengthy definition that includes a checklist of specific requirements that OSS licenses must meet. The OSI lists licenses that they have determined meet their criteria. The Free Software Foundation (FSF) (http://www.fsf.org) publishes the "Free Software Definition," which defines the related term "Free software" (a term they use to indicate user freedoms, not no-cost). The key is that OSS is licensed in a way that grants users various freedoms, including the ability to make improvements and to collaborate with others when creating these improvements. Software that is not open source is often called "proprietary software" or "closed source software." As explained below, nearly all OSS is commercial software as defined in U.S. law.

B. How is OSS typically developed?

OSS is typically developed collaboratively with users and other developers as part of an OSS project. Figure 1 presents a simplified model of how OSS is typically developed. Users can get the software from a location termed a "trusted repository"; this is typically a publicly accessible website, possibly running on top of common sites such as GitHub (https://www.github.com) or SourceForge (http://sourceforge.net). Users can also get the software indirectly via distributors, who may package the software with other software or services (such as warranties and training). Some people are authorized to make changes to the trusted repository; these people are trusted developers. Users can submit bug reports and feature requests to these developers, typically through the OSS

project's website. All of these processes are not significantly different from those for proprietary (a.k.a. closed source) software.

A key difference is that OSS users need not merely be consumers of the software. Users can also modify the software or hire someone to modify the software. Users can choose to keep those software modifications inside their organization, but this is often an unwise decision. Keeping modifications within an organization can be costly because this can make it difficult and expensive to update to the next version of the software (since these modifications must be re-merged). Software sustainment costs are typically 80% of the software lifecycle, so choosing to maintain a modified version "in-house" can quickly escalate an organization's costs. An obvious solution is to contribute those changes back to the OSS project's trusted developers, eliminating the need to re-merge the changes. If a user continues to do this, the user may eventually become a trusted developer. This collaboration, over time, can lead to a vibrant community of developers and users of that OSS.

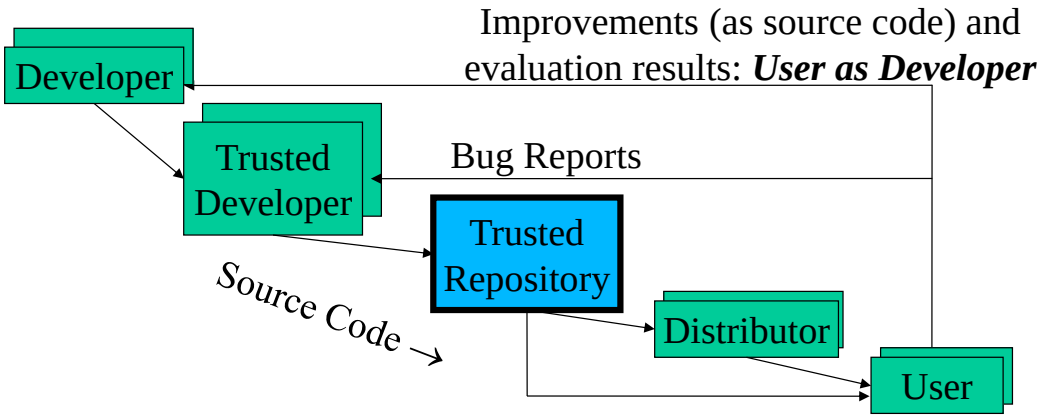

Figure 1. Model of Open Source Software Development Process

C. What are examples of successful OSS projects?

Well-known successful OSS projects include the Linux kernel (https://www.kernel.org) and Apache web server (httpd.apache.org). Examples of successful OSS projects that are transitions of government-funded research and development include the Internet (whose TCP/IP protocols were developed with government funds and released as OSS), Suricata (suricata-ids.org), Security-Enhanced Linux (SELinux) (selinuxproject.org), and OpenStack (http://www.openstack.org/). For a much longer list of government-funded OSS work (including lists of lists), see http://www.dwheeler.com/government-oss-released/.

D. Why use an OSS approach?

There are many potential advantages for using an OSS approach. The Department of Defense (DoD) has identified many positive aspects of OSS, particularly for its users. In particular, "the continuous and broad peer-review enabled by publicly available source code supports software reliability and security efforts through the identification and elimination of defects that might otherwise go unrecognized by a more limited core development team. The unrestricted ability to modify software source code enables [users] to respond more rapidly to changing situations, missions, and future threats…OSS is particularly suitable for rapid prototyping and experimentation, where the ability to 'test drive' the software with minimal costs and administrative delays can be important." [DoD OSS 2009]. Many quantitative studies show that OSS programs are often a reasonable or superior alternative to their proprietary competition according to measures such as market share, reliability, performance, scalability, security, and total cost of ownership [Wheeler2014a].

E. What are some common misconceptions about OSS?

OSS is often mistakenly called "non-commercial." OSS almost always meets the definition of a "commercial item," as that term is defined under U.S. procurement law (41 U.S.C. § 103(1)). Under this definition, an item qualifies as a "commercial item" if it is not real estate, is customarily used for non-government purposes, and is either licensed to the public or offered for license to the public. Since nearly all OSS meets these criteria, nearly all OSS can be considered commercial computer software.

When dealing with the government, it is important to understand that OSS is almost always commercial, for at least three reasons. First, government officials must follow a large set of rules; if they do not know what rules to follow, they often find it difficult to act. Once they understand that OSS is usually commercial, this problem disappears, because they are typically already familiar with the rules for commercial software. Second, U.S. Government procurements must give commercial items (including nearly all OSS) appropriate statutory preference in accordance with 10 USC 2377 and include them in market research. For more information, see the Federal Acquisition Regulations (FAR) 2.101(b), 12.000, and 12.101, and the DoD FAR Supplements (DFARS) 212.212 and 252.227-7014(a)(1). Third, once government officials and contractors understand the statutory definitions and requirements, they realize that they are required to consider OSS and that releasing OSS is a valid commercialization approach.

Some organizations think that they cannot develop OSS if they also develop proprietary software, or intend to develop proprietary software in the future; however, this is not true. Many organizations participate in the development of OSS while simultaneously developing proprietary software, or they use OSS as a portion of a proprietary product.

Software released to the public as OSS can be used and modified by anyone, and the original developers retain fewer or no exclusive rights to it. This enables all researchers (not just the original developers) and users to collaborate together. Enabling user-centered innovation can offer great advantages [vonHippel], and enabling a wider pool of researchers has the potential of speeding research as well. Organizations are free to use the OSS they develop to develop products and services, subject to all legal and contract requirements.

Some mistakenly assume that OSS is always less secure than proprietary software. The DoD notes that, "The continuous and broad peer-review enabled by publicly available source code supports software reliability and security efforts through the identification and elimination of defects that might otherwise go unrecognized by a more limited core development team" [DoD2009]. In practice, some OSS is very secure, and some is not, so individual OSS programs must be evaluated on their own merits. There is a related myth that anyone in the world can change OSS and this immediately changes the software in user supply chains. In actuality, only some (trusted) developers have the permissions to change any particular OSS program, and the distributed nature of its source code makes it much easier to detect unauthorized changes. In addition, users can choose which software and supply chain to use. Users must always evaluate the software they use, whether it is OSS or not, to determine whether it is appropriate for their purposes.

An OSS approach is different from other approaches for developing or procuring software that government personnel and contractors are often more familiar with. [Wheeler2013b] discusses some of the key challenges and opportunities in the government application of OSS.

F. What is legally required to release government-funded OSS?

Software can only be released as OSS by those who have the legal right to do so. [Wheeler2011a] includes a list of five main questions to answer to determine whether someone (identified as "you") has the legal right to release government-funded software as OSS:

1. What contract applies (including terms and decisions)?
2. Do you have the necessary copyright-related rights?
3. Do you have the other intellectual rights (e.g., patents)?
4. Do you have permission to release to the public?
5. Do you have all the materials (source code) and are they properly marked?

If an OSS approach is to be used, it is best to ensure that ensure that all legal requirements will be met *before* any code is written.

4. ABCs of OSS Approaches to Technology Transition

As noted earlier in section C, the "Need, Approach, Benefit, and Competition" (NABC) framework is a simple way to quickly communicate an idea and its value proposition [Carlson2006] [SRI2011] [Schindlholzer2008].

Here we briefly describe a potential NABC description for using an OSS approach as a whole to technology transition in general. This may be useful when considering whether or not to apply an OSS approach in a particular case.

- *Need*. The need for research work is specific to the research work itself. In U.S. cybersecurity research, the rationale can typically trace back to the fact that the United States depends on computer and network systems, and those are under attack.
- *Approach*. The general approach for applying OSS is described in section 3. When developing software under research it is possible to release OSS in various ways; for our purposes we will focus on these three:
  - *Released as unmaintained software*. Here a typical primary goal is to document what was done for reproducibility (a vital requirement for science that is often lost) and demonstration, instead of software that will be used directly. We recommend in these cases that it be designed to be easy to automatically rebuild (e.g., include ant, maven, cmake, automake, or makefile directives) so that the results can be reproduced more easily.
  - *Modify existing OSS*. Here a typical primary goal is to get the software functionality into people's hands, and there is an existing OSS project that can be modified to do so. Where the OSS is already in wide use, this makes it easy to get the functionality out to others, but the PM must work with the existing OSS project to ensure that the new functionality will actually be included.
  - *Start new OSS project*. A typical primary goal here is to get the software functionality into people's hands, and there is no existing OSS project that can be modified to do so (or there are problems doing so). When starting a new project there's no requirement to work within an existing structure, but it will take time to ensure that potential users can actually find the software, never mind use it, and it will take time to enlist new co-developers.

- *Benefits of OSS approach*. Section D discusses some of the advantages of OSS, particularly from a user's point of view. For a PM and researcher, an advantage is that the idea can be quickly adopted and deployed. Another advantage is that the researchers can collaborate with users of the software, enabling user-centered innovations that offer "great advantages over the manufacturer-centric innovation development systems" [vonHippel].
- *Competition for OSS approaches*. Some alternatives to OSS release are:
  - *Paper publication alone.* Many academics only publish papers (especially if they are in a "publish or perish" environment). However, papers omit vital information that can lead to irreproducibility. The experience of the LIMMAT developers is an instructive example: "From the publications alone, without access to the source code, various details were still unclear…what we did not realize, and which hardly could be deduced from the literature, was [an optimization] employed in [the previous work] GRASP and CHAFF [was critically important]…Only [when CHAFF's source code became available did] our unfortunate design decision became clear…The lesson learned is, that important details are often omitted in publications and can only be extracted from source code. It can be argued, that making source code… available is as important to the advancement of the field as publications" [Biere2004]. What's more, paper publications cannot be executed, and thus cannot be tested, examined, or improved in the ways that executable software can. If the goal is adoption of an approach, "running code" is more likely to be effective than merely publishing a paper.
  - *Proprietary software or service.* Another approach is to work with researchers to create proprietary software or a service, and sell that. This approach can work and is well-understood, and DHS has done this many times. However, the disadvantages of this approach are often not as widely understood, and justify considering alternatives. From the government PM's point of view, establishing a proprietary software or service as the result of research often creates a situation where there is only a single effective supplier (especially if the researchers gain exclusive rights to any patents or copyrights for the works created using government funds). Monopolies tend to drive costs up and quality down for potential users. Should the business fail, the results may become unavailable to everyone. Proprietary software and services are often difficult to build on, and thus, this approach can actually impede further research in the same area (those who control the proprietary software or service can decide who make improvements, and how, and it may not be in their interests to encourage

this). Also, this approach typically requires establishing a business; many researchers are not interested in, or not good at, establishing and running a business. External businesses might buy rights, but in many cases existing businesses will have some difficulty evaluating the work and turning it into a product. External businesses may have difficulty turning the approach into a product, or may even choose to purchase exclusive rights to prevent their use (to inhibit competition and thus raise their profits). Finally, these approaches tend to inhibit, instead of enable, user-centered innovation; a proprietary business earns money because it has the exclusive right to maintain the software, and this exclusive right can be threatened by user-centered innovation. Proprietary software can bring innovations to users, but these potential issues suggest that other avenues should be considered.

The following figure illustrates these three different ways to release software as OSS in the context of research work.

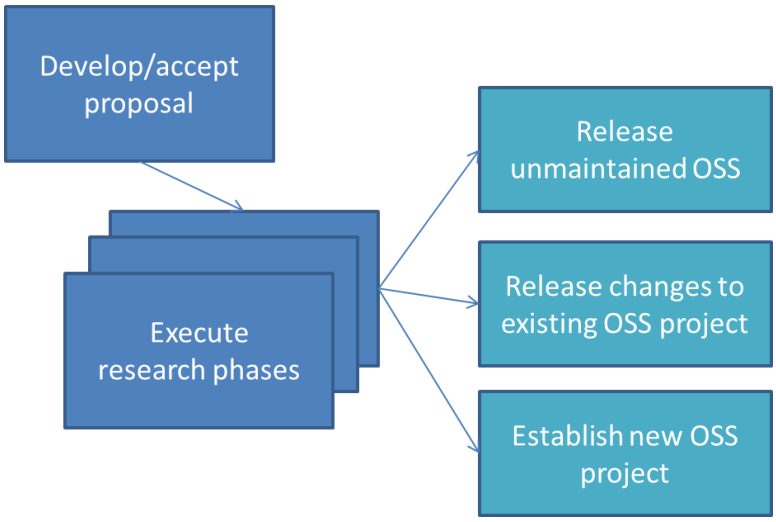

Figure 2. Options for Releasing OSS from Research

If the research releases unmaintained OSS, it is possible that someone else may use it to start an active OSS project. In the other cases, the research works with an existing OSS project or starts a new OSS project. In all cases, once the software is added to an existing OSS project or is used to start a new OSS project, there is interaction with the users that is vital to its future use.

The following figure shows how an OSS project can interact with its users, focusing on some of its inputs and outputs and the issues related to technology transfer. An OSS project performs development and sustainment, governed by some sort of governance and licensing process. The OSS project takes in various inputs, including contributions and

feedback, reused components, and standards. It produces a project website (where people can go to get the software or contribute to it), as well as the software and documentation itself. Ideally it should go to users and potential users, but many impediments may interfere with this (as we will examine and discuss how to counter). Ideally the users and potential users provide contributions and feedback, leading to improvements.

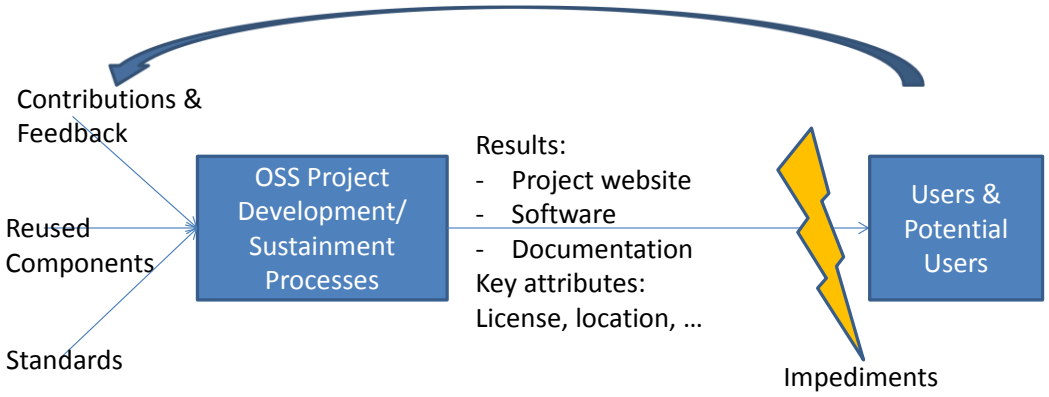

Figure 3. OSS Project Interactions

The following sections of the document discuss these issues, focusing on how to increase the likelihood of success. We will apply the NABC framework more minutely in several cases to identify various more detailed ways to apply an OSS approach.

5. Building on Existing Work

Research should build on existing work where practical. Identifying relevant existing OSS projects early can eliminate a lot of unnecessary work (by avoiding redevelopment of software that already exists). It will also speed up later technology transition, because the researchers will have thought about how to deliver the improvements to potential users and possibly embedded the improvements into software that is already in use.

A. Evaluating existing software

Researchers should identify various reuse options, analyze them, and then select the major software components that they plan to use, extend (e.g., through modifications or plug-ins), and create (This list of options is sometimes called "adopt, modify, and create."). This should be done prior to designing and implementing any software, since it becomes increasingly more difficult to take advantage of this information later.

We recommend that the researcher document this information in a short *reuse options report*. In the report the researcher identifies the major component candidates found, along with the key features of each. The report should also identify the final selections and the rationale for those selections. This basic analysis of alternatives should justify the final choices. If a choice turns out to not work for some reason, the report can reduce research risks because alternatives have been documented. The report also provides confidence to others (such as PMs) that the key options have been considered. The report need not be lengthy; it simply needs to identify the key alternatives, key traits, and final selection. The process of writing the report increases the likelihood that important reuse options will be considered.

The analysis process involves evaluating different software components. There are many ways to evaluate software. One way is to use the four-step "IRCA" process described in [Wheeler2011b]: (1) ***I***dentify candidates, (2) ***R***ead existing reviews, (3) ***C***ompare the leading programs' basic attributes to the needs, and then (4) ***A***nalyze the top candidates in more depth. The following paragraphs describe this, and list specific issues relating to using OSS to support cybersecurity technology transition.

The first step in the IRCA process is to *identify* candidate software for reuse. Search engines such as Google can often find OSS software components that have been given search terms for the functionality. The HOST project website at https://host-project.org/ provides a list of many other potentially useful components, in particular, its "Open

Security Catalog" at https://host-project.org/open-security-catalog. [Wheeler2011b] provides some more tips for finding relevant candidates. A central repository of components for a programming language or an operating system being used can also be an excellent source (see section D below). Some examples of common tasks in cybersecurity-related research, and plausible existing projects that support them, include the following:

- Static source code analysis: LLVM/clang (http://llvm.org/), gcc (https://gcc.gnu.org/), Frama-C/why3 (http://frama-c.com/ and http://why3.lri.fr/)
- Dynamic web application analysis: W3AF (http://w3af.org/), OWASP ZAP (https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project)
- Packet analysis and network intrusion detection/prevention: Wireshark (https://www.wireshark.org/), Suricata (http://suricata-ids.org/)
- Satisfiability modulo theories (SMT) module: CVC4 (http://cvc4.cs.nyu.edu/web/), alt-ergo (http://alt-ergo.lri.fr/).
- Boolean satisfiability (SAT) solver: MiniSAT (http://minisat.se) and Sat4j (http://www.sat4j.org/),.
- Fuzz testing: American Fuzzy Lop (AFL).

Reading reviews, *comparing*, and *analyzing* are the next steps. For the most part, they are the same for any software, whether it is OSS or not. As always, important attributes to consider include functionality, cost (including costs to make changes), market share, support, maintenance, reliability, performance, scalability, usability, security, flexibility/customizability, interoperability, and legal/license issues.

In some cases existing software should *not* be reused. In some cases a reused component may be hard to build or install on a user's system. Also, a reused component with vulnerabilities may insert vulnerabilities into your project. In particular, if only a very small portion of a project is reused, or if using the component will cause problems for users, perhaps that small component should be broken out, something else should be considered, or that small functionality should be redeveloped.

Some plausible candidates for reuse might be OSS, and others might not be. We will first discuss the OSS case, followed by the non-OSS case.

Researchers should be sure to check each OSS project to verify that it actually has an OSS license. The OSI maintains a list of licenses that they have certified as OSS licenses at http://opensource.org/licenses. Unfortunately, some software projects fail to include any license. Generally speaking, the absence of a license means that the default copyright laws apply. This means that the developer retains all rights to the source code

and that nobody else may reproduce, distribute, or create derivative works. [Github2014].

Researchers considering OSS components should prefer OSS projects that are active (e.g., ones that have active discussions through mechanisms like mailing lists and routinely respond to reported issues). The majority of successful growth projects have greater than 1,000 downloads [Schweik2012,272], so projects with many downloads should be preferred. Obviously, ones that provide necessary functionality and quality should be preferred. Quality can be difficult to measure directly, but there are often indirect measures that provide evidence of quality (e.g., many users and a strong regression test suite). However, note that missing some functionality is not necessarily a problem; in some cases, it may be better to create the missing functionality for an otherwise-useful OSS program.

Sometimes promising non-OSS components can be reused to build or run a new OSS component. However, if these non-OSS components are required, it is important to examine their potential disadvantages (as well as their advantages) to make a reasoned decision.

It is typically more difficult to deploy an OSS component if non-OSS components are *required* to build or use it. Since the whole point of an OSS approach is to maximize potential collaboration, anything that impedes that collaboration can inhibit further development and thus increase the risk of long-term failure. It is fine to *port* (run) on non-OSS components, as long as there is a way to use an alternative OSS component; indeed, porting to *optional* non-OSS components can increase the likelihood of collaboration.

Some non-OSS components are widely ubiquitous and cheap enough that deployment is not as significantly impeded. Researchers should state this (noting any exceptions). PMs should examine the list of required non-OSS components to evaluate the potential risks (if any) that might be caused by limited collaboration.

If an existing OSS component is being considered for use but must be modified, investigate early any particular contribution requirements. In particular, check whether it requires any contributor agreements or contributor (copyright) assignments; these may require significant legal review, and this review can cause delays or prevent use of the component.

B. Modifying existing OSS

After examining the options, it may be determined that the best way forward is to modify some existing OSS to provide new functionality or capacity. If these modifications are necessary to deploy the new capability, and the intent is not to simply throw away the resulting software, then an effort should be made to submit those changes

to existing OSS projects in a way that the project can accept. This avoids "project forking" (creating a separate derivative project) that has to be separately maintained. A project fork will not normally be maintained by those collaborating on the original project and thus risks becoming obsolete or imposing higher (duplicative) maintenance costs.

It is not wise to commit, in a contract, that someone else will accept some unwritten proposed changes. However, it is possible to commit to submitting the proposed changes. Also, a little planning and early discussion usually prevents later problems.

Most OSS projects have a number of rules for proposed changes, and an effort should be made to meet them. Discussions with the OSS project developers should take place before changes are made to determine these rules and a plan to meet them. For example:

1. OSS projects typically require that proposed changes be submitted using the project's OSS license or a license that is even less restrictive. Otherwise the project would have to change its license to accept the change, and this is unlikely. It is possible to release changes under other licenses also.
2. If the changes are large, an OSS project will typically require the changes to be broken into smaller logical changes so that they can be separately reviewed for correctness. Changes are usually tracked using version control software such as git, mercurial, or subversion.
3. The proposed changes must be easily compiled, tested, and run in an automated way using widely available tools and libraries (preferably those already used by the project).
4. Proposed changes should enable easy and separate updates of dependencies such as support libraries. If a proposed change embeds dependencies (such as libraries) in a way that makes them difficult to update, those dependencies can quickly become obsolete.

There are many ways to contribute proposed changes to an OSS project. Proposed changes may be provided in a text file or an email body using the patch command's format; proposed changes in this form are often called "patches." Users of distributed version control systems, including the widely-used git program, may instead send "pull requests." For consistency, we refer to these proposed changes (regardless of their form) as "changes."

Potential submitters of changes must go to the OSS project website to determine how the project prefers changes to be submitted, and then follow the project's preferred approach for submission and responding to reviews. In general, researchers should work with the OSS project developers and try to make accepting changes easy; many OSS

projects have a large number of proposed changes to consider. OSS projects vary in many ways on the details of how they accept and process changes; however, there are many nearly universal rules that researchers should follow; here are some of them ([Raymond2000] [Kegel2004]):

1. Follow the OSS project's conventions (e.g., choice of language, style guide, version control system, submission process).
2. Send changes against a current version of the software. If the changes are against an old version, they can be difficult to integrate. If the researcher only has changes against old versions of the software, update and apply the changes to a current version first.
3. Include just one bugfix or new feature per change. Do not send a massive single change, since these are nearly impossible to reasonably review. This is especially easy to do using distributed version control software such as git.
4. Make any code changes easy to read. Carefully choose names (especially in interfaces), include comments that define every application programming interface (API) (e.g., every class, method, function, or procedure), and include comments to explain unusual or complex situations that cannot be simplified.
5. Include regression test cases. Adding test cases for a bugfix or new feature when the bugfix or feature is added makes it far less likely that future changes will cause undetected failures.
6. Be respectful to people, and hard on the code. In many cases, the OSS project will request the researcher to revise the proposed changes before they will be accepted (because of defects, design limitations, style problems, and so on). Researchers should respond to those requests and re-submit the updated changes.

C. Standards (de jure and de facto)

There are many data format and interface standards. Using standards (particularly those for data formats and interfaces) can make a component much easier to integrate into larger systems, simplifying technology transition. However, some standards are released, or have patents enforced, in a way that discriminates against some uses (such as use by OSS). Thus, although use of standards is generally encouraged, standards use should be carefully weighed if they discriminate against those who wish to voluntarily collaborate.

Thus, researchers should identify ahead of time any relevant standards (for use, improvement, or creation), and check to see whether these standards are actually compatible with their technology transition approach. A significant amount of literature on this topic and about the definition of the term "open standard" is available. For our

purposes, a useful start is OSI "Open Standards Requirement (OSR) for Software" (http://opensource.org/osr). The OSR's fundamental requirement is that an open standard "must not prohibit conforming implementations in open source software" and has the following criteria (capitalization is retained from the original):

1. "No Intentional Secrets: The standard MUST NOT withhold any detail necessary for interoperable implementation. As flaws are inevitable, the standard MUST define a process for fixing flaws identified during implementation and interoperability testing and to incorporate said changes into a revised version or superseding version of the standard to be released under terms that do not violate the OSR.
2. Availability: The standard MUST be freely and publicly available (e.g., from a stable web site) under royalty-free terms at reasonable and non-discriminatory cost.
3. Patents: All patents essential to implementation of the standard MUST:
    a. be licensed under royalty-free terms for unrestricted use, or
    b. be covered by a promise of non-assertion when practiced by open source software
4. No Agreements: There MUST NOT be any requirement for execution of a license agreement, non-disclosure agreement (NDA), grant, click-through, or any other form of paperwork to deploy conforming implementations of the standard.
5. No OSR-Incompatible Dependencies: Implementation of the standard MUST NOT require any other technology that fails to meet the criteria of this Requirement."

For this document, a standard is a specification that has achieved some sort of consensus, regardless of its source or history. There are many standards-setting bodies, such as the Internet Engineering Task Force (IETF) and the International Organization for Standardization (ISO). Country-based organizations, such as the National Institute of Standards and Technology (NIST), also develop or advance standards. Standards may also be developed by smaller or less formal groups; this is especially likely for highly specialized fields or uses. For example, Satisfiability Modulo Theories (SMT) tools automatically determine the satisfiability of first-order mathematical formulas with respect to some logical theory. Some defensive cybersecurity tools may be productively implemented using SMT tools, or enhanced by improvements to SMT tools. In these cases, the researcher should consider using the SMT-LIB (http://smt-lib.org/) standards, since using them can simplify replacing one SMT implementation with another. Similarly, those using automated theorem-proving (ATP) tools should consider using the

Thousands of Problems for Theorem Provers (TPTP) language, since many ATP tools support this language (http://www.tptp.org).

A standard may be formally approved by some standards-setting body, but not actually in use. Thus, before committing to a standard, researchers should determine whether it already has users and what the alternatives are. It may be fine to use a newly released standard or re-invigorate a standard that has fallen into disuse, but researchers should determine where this is the case and consider how to reduce risk where they can.

Researchers must test compliance with key open standards; otherwise, in practice, software will not comply. Where possible, this should be demonstrated by replacing each component that implements an open standard with an independent implementation that also implements the open standard. For example, if a web application is built on open standards such as HTTP, HTML, and CSS, it must be demonstrated with alternative common web browsers that implement those standards. [Scott2011]

If a new standard must be created (which itself will typically build on existing work), consider having it developed or eventually maintained by an independent standards-setting body. It may be best to work with organizations that will make the specification available free of charge, since such specifications are more likely to be read (and thus used). Some standards-setting organizations that can support this include the IETF, W3C, and OASIS. Some organizations (such as ISO) typically charge large fees for copies of their standards, even though they often do *not* pay the authors and technical reviewers for their work. Historically, this was justified because of the costs of printing. However, this practice has become controversial; the Internet has eliminated this justification, and the fees have become a serious impediment to the use of standards since modern systems build on a large number of standards [Jelliffe] [Weird2010]. Governments often pay for the development of standards using public funds; in these cases it is becomes especially difficult to justify why the public must pay again to receive the standards they already paid to have developed. It is possible to get organizations like ISO to allow standards to be published free of charge (this occurred with the Common Criteria), but this must typically be agreed on ahead of time. Researchers who want to ensure that standards are widely available to potential users may prefer developing standards in an organization that has a practice of releasing the document to the public without fee (such as IETF, W3C, and OASIS), or get a special agreement in writing.

6. Establishing New OSS Projects' Collaborative Environments

In some cases it's necessary to start a new project, whether or not some existing OSS can be reused. In these cases, there are other issues to consider, as this and the following sections discuss. Knowing about these issues can also help when examining existing OSS projects, or when considering a major change in an existing OSS project's governance or license.

In this section we consider the basic issues that must be considered for establishing a collaborative environment (an environment in which collaboration can take place). This includes choosing a name and purpose, a hosting service (a technical infrastructure for collaboration) such as GitHub, a foundation if one will be used, a governance process, and a license, and deciding whether any contributor agreements or assignments will be used. For most of the issues we discuss the need, potential approaches, benefits, and competition (NABC).

A. Project name and purpose

Researchers should give each new project a name that is distinct, clearly pronounceable, and easily found via a web search. Do not use common words like "the" or "why" as a name. It is often best to choose a unique name that is not used by another project (Google searches and examining large repositories like Debian's can increase the likelihood of choosing a name that is unique). A unique name may make the name harder to pronounce, but a unique name makes the project easier to find… and that is typically more important.

Each new project needs a concrete, limiting, and short purpose statement so that others will understand what the project is about; craft this statement so others will understand it. The statement should clearly *differentiate* the project from other projects. For example, "improve the world" is not an appropriate purpose; it is too vague and is not a differentiator from other projects (most software is written to improve the world in some way). In contrast, "develop a high-speed web server" is a clear purpose (if most people in your audience know what a "web server" is).

B. Hosting Services

Researchers starting a new project need to establish basic technical mechanisms for collaboration. Today this would normally include a version-controlled public code

archive for the software (usually managed using git), an "on-ramp" web page and supporting material so potentially interested people can learn about the project (this includes basic information about the project's goals), an issue tracker for bugs (defects) and feature requests, a mechanism for group discussion (often a mailing list), and a way to release new formal versions. Many projects include a wiki for easy contribution of new information (this wiki may allow anyone to edit or may restrict who can directly make changes). There are many options and technical approaches for doing this, and we expect even more options to come over time. At this time we recommend using git for version control of a new project; it is OSS, widely used, widely supported, and supports a number of mechanisms that aid collaboration. Researchers may choose to simply use an existing hosting service (such as GitHub), or may establish their own.

These technical services are available from many hosting services (sometimes called "forges"), and are often free for OSS development. GitHub is an especially popular choice today. Many other hosting services exist, including SourceForge, Atlassian Bitbucket, GitLab, GNU Savannah, and Canonical's Launchpad.net[2]. Note that GitHub and SourceForge can be especially convenient for U.S. federally funded work, because they have existing agreements with the U.S. Government on their terms of service for U.S. Government employees.[3]

It is also possible to self-host a hosting service (on one's own systems or some cloud service), though this is typically not a good approach for a new OSS project. OSS options for self-hosting software include Allura (the Apache Foundation software that runs SourceForge), GitLab, Kallithea, and Savane (the software that runs Savannah). Git itself is OSS, but the GitHub software is proprietary; GitHub's proprietary software can be separately licensed for self-hosting as "GitHub Enterprise." GitLab's "GitLab Community Edition" is OSS; they also sell a proprietary variant of it for self-hosting called GitLab Enterprise. Self-hosting is common for proprietary software development, in part to reduce the risks of exfiltration, but since OSS software is normally available to the public, this concern is irrelevant. Many larger OSS projects self-host, to support their various special needs, but when starting a new OSS project it is often unclear what special needs might apply. Thus, it is typically better to use some existing hosting service when starting a new project and then consider self-hosting once the project is more established and its special needs (if any) become clear. Modern hosting services now provide enough services and "hooks" that even special needs often don't require

---

[2] A much longer list of hosting options can be found at https://en.wikipedia.org/wiki/Comparison_of_source_code_hosting_facilities.

[3] GitHub is also used for collaborative development of government-related documents, e.g., https://github.com/WhiteHouse/fitara/pull/39.

self-hosting today. In short, while self-hosting is a possible approach, in most cases this is not the best approach for a new project.

Changing the hosting service can take time, so it is best to choose a service that will meet expected needs. Data directly managed by git is often relatively easy to move because git is a distributed version control system (DVCS); this means git can easily make a copy of a repository and put it somewhere else. However, it may be harder to move other information (such as tracker data and wiki contents) and mailing lists. In addition, it may take time to adjust processes when moving to another hosting service. The time and effort necessary to move to another site will temporarily detract and slow project progress. Perhaps most importantly, users over time will expect a URL to be the "official" project site, so for a time a new site will have a lower search rank and be harder for users to find.

C. Foundations

Researchers may choose to create an OSS project under the umbrella of a larger organization, sometimes called a "foundation." Foundations provide a variety of services and mechanisms, depending on the foundation. These often include a formal legal framework (often as a non-profit), as well as already established mechanisms for governance, conferences, and marketing. In many cases foundations can accept money and manage that money for the project. Foundations can be very helpful when starting a new project, especially if the new project is strongly related to an existing project in the foundation, or if the researchers are not familiar with OSS development processes.

However, foundations typically impose a number of additional rules on their member projects (e.g., licensing requirements, legal processes, and so on). For example, some foundations run their own hosting service, or have a preferred hosting service, so foundation and hosting service decisions may be interconnected. Foundations often impose specific governance structures, which may or may not be desired. In addition, it can be more difficult for a project to leave a foundation than to join it. Therefore, researchers should make sure that a foundation is compatible with the researcher's goals and technology transition plan before choosing to create a new project within a foundation. A well-chosen foundation can reduce risk; a poorly chosen one can increase it.

Some well-known organizations that operate as foundations include:

- Apache Software Foundation (ASF) (http://www.apache.org)
- Eclipse Foundation (http://www.eclipse.org)
- Free Software Foundation (FSF) (http://www.fsf.org)
- Software in the Public Interest (SPI) (http://www.spi-inc.org)

- Open Web Application Security Project (OWASP) (https://www.owasp.org)
- Linux Foundation (LF) (http://www.linuxfoundation.org/).

In cybersecurity research OWASP is a common choice when choosing a foundation, but it is by no means the only one.

Most projects do not choose a foundation, at least initially, unless their work is strongly connected to an existing project in an existing foundation. We recommend that researchers consider using a foundation, but not start with one unless the foundation provides specific advantages to the specific project; it is always possible to join one later.

D. Governance (decision) process

An OSS project needs to encourage collaboration, but it must also have a way to make decisions (such as rejecting contributions where warranted). Common governance approaches for making final decisions include a *benevolent dictator* (one person who is in charge of the final decision) and "group decision-making" (where a group votes). Governance can change over time, but it is much easier to change once there is some governance process in place. After all, once there is a process for making decisions, that process can be used to change how to make decisions in the future.

The job of those in governance is to encourage improvements to the project's results, including their functionality, quality, and documentation. Governance is needed because not all proposed changes are actually improvements. First-time contributors, in particular, often need information on how to modify their proposed changes so that they will work in a larger context.

Initial projects typically have a benevolent dictator because this is easy to establish and execute, so we recommend that in most cases as a starting point. Most starting projects do not need a more complex decision-making process. However, if multiple parties already have a stake in the result, a different governance process that explicitly lets them have a voice may be needed.

The benevolent dictator model is not as dictatorial as it sounds. All OSS licenses permit the creation of *project forks* – that is, competing projects based on a version of the pre-existing project's source code. This creates a built-in escape valve if the governance process becomes actively harmful to the project. As stated in [Wheeler2014], "The ability to create a [project] fork is important in FLOSS development, for the same reason that the ability to call for a vote of no confidence or a labor strike is important. Fundamentally, the ability to create a fork forces project leaders to pay attention to their constituencies…Often, the threat of a fork is enough to cause project leaders to pay attention to some issues they had ignored before, should those issues actually be

important. In the end, forking is an escape valve that allows those who are dissatisfied with the project's current leadership to show whether or not their alternative is better."

E. Licenses for new software

If you are modifying existing OSS, you would typically use at least the existing license. That way, the modifications can be incorporated into the main project. You may in some cases also release your changes with other licenses as well. However, with new projects, an important question is what license to use.

Choosing a license for a new OSS project is an important step. A license determines the rules of the road for potential contributors, and it is sometimes difficult to change after the proposed work is started.

A complication of license selection is that there are different options, each with different pros and cons. However, some basics can be stated up front. The following are important requirements researchers should consider when choosing an OSS license for a new project (based on the DoD OSS FAQ [DoDFAQ]):

1. *Do not create a new OSS license*. New licenses often fail to be OSS, even if they are intended to be, because few lawyers are trained in how to create OSS licenses. Even if a new license is OSS, a new license (typically called a "vanity license") imposes a large legal burden on every user, developer, and organization, each of which must examine the license text to understand its terms and check its compatibility with other licenses.
2. *Choose an already accepted OSS license.* Choose a software license that is recognized as an Open Source Software license by the Open Source Initiative (OSI), is also recognized as a Free Software license by the Free Software Foundation (FSF), and in addition is acceptable to widely used Linux distributions (such as being a good license for Fedora). License evaluation is a difficult legal specialty; it is much better to choose licenses that have already been evaluated by specialists. The licenses recommended in the following paragraphs all pass these tests.
3. *Ensure the license will work with the expected uses.* Ensure that the license will permit (or even enable) anticipated uses and business models. If the OSS component must work with other components, or is anticipated to work with other components, ensure that the license will permit it. In particular, if the intent is to embed the software into an existing OSS component, choosing that component's license or at least a license compatible with it is typically a wise choice.

4. *Choose a GPL-compatible license*. The GNU General Public License (GPL) is the most common OSS license. There is no need to necessarily use the GPL, but it is usually unwise to choose a license incompatible with the majority of OSS licenses. In particular, avoid releasing software only under the original (4-clause) BSD license (which has been replaced by the new or revised 3-clause license), the Academic Free License (AFL), the now-abandoned Common Public License 1.0 (CPL), the Open Software License (OSL), or the Mozilla Public License version 1.1 (MPL version 1.1, which has been superseded by the GPL-compatible MPL 2.0). The Eclipse license is also unfortunately GPL-incompatible.

   The Apache 2.0 license is incompatible with GPL version 2, but is compatible with GPL version 3, and most software released under the GPL is released as "version 2 or later" (much of the rest is version 3 or versions 2 or 3). Thus, for our purposes we will consider the Apache 2.0 license as generally GPL-compatible.

5. *Choose a popular license*. Unusual licenses can inhibit collaboration in at least two ways. First, every organization must separately review each new license to understand its ramifications. This additional legal review effort, which often must be done by a lawyer in each organization that is considering the software, can significantly slow any initial use or acceptance of the software. Second, unusual licenses are often incompatible with other more popular licenses, resulting in an inability to use the software in important ways (and thus discouraging its use and support). The OSI maintains a list of what it perceives as "popular licenses" at http://opensource.org/licenses.

Given these basic requirements, researchers must choose a license that best meets their goals. A reasonable approach is to pick the type of OSS license (from three basic types), then select a license of that type after learning about the popular options for that type. Here are the three basic types of OSS licenses, along with recommended popular OSS licenses for each choice:

1. *Permissive*. A permissive license permits arbitrary use of the program, including making proprietary versions of it. If the goal is to maximize the use of a technology or a standard in a variety of different applications/implementations, including proprietary software, then permissive licenses may be especially useful. Popular permissive OSS licenses include the **MIT license**, the **BSD 2-clause** (Simplified or FreeBSD) **license**, the **BSD 3-Clause** (New or Revised) **license**, and the **Apache license 2.0**. The Apache license 2.0 includes statements that provide some protection for later developers and users against patent claims by previous contributors.

2. *Strongly protective*. A strongly protective license protects the software from becoming proprietary, and instead enforces a "share and share alike" approach between parties. Strongly protective licenses even forbid linking the software into a larger proprietary work. If the goal is to encourage longevity and cost savings through a commonly maintained application, protective licenses may have some advantages because they encourage developers to contribute their improvements back into a single common project. Such licenses do prevent linking the software into proprietary software, but the software can still be used in many other ways. In particular, like all OSS licenses, the GPL can be used for any purpose (including commercial purposes) and such software can be commercially supported by one or more organizations. The most popular strongly protective OSS licenses are GNU **General Public License (GPL)** version 2 or 3. Typically these are implemented as "GPL version 2 or greater" or "GPL version 3 or greater" to enable future compatibility.

3. *Weakly protective*. A weakly protective license is a compromise between the two previous categories (permissive and strongly protective), preventing the covered component (typically a library) from becoming proprietary yet permitting it to be embedded in larger proprietary works. If the goal is to encourage longevity and cost savings through a commonly maintained library while allowing the library to be included in a larger proprietary work, then weakly protective licenses may have some advantages. The most popular weakly protective licenses are the GNU **Library or Lesser General Public License (LGPL)** version 2.1 or 3. Typically these are implemented as "LGPL version 2.1 or greater" or "LGPL version 3 or greater" to enable future compatibility. The LGPL or GPL may including a "GPL linking exception" that enables software projects which provide library code to be "linked to" the programs that use them, without applying the full terms of the license to the using program under certain conditions.

The strongly protective and weakly protective licenses are sometimes also called copylefting licenses.

When choosing a license, researchers should examine the major options and implications. Many documents and web pages discuss OSS licensing and its implications; read several of them to get a fuller picture. GitHub provides the web site http://choosealicense.com/. OSS Watch provides an on-line license differentiator at http://oss-watch.ac.uk/apps/licdiff/. An actively-maintained Wikipedia page compares OSS licenses at https://en.wikipedia.org/wiki/Comparison_of_free_and_open-source_software_licenses. Other documents, such as [Morin2012], can also be helpful. Finally, read the actual text of the most promising licenses; none of them are long (for legal documents), and in the end it is that text that governs.

The GPL is the single most popular OSS license. However, a large amount of OSS software is released under permissive licenses, and the total of the permissive licenses is greater than that of the protective licenses. The popularity of various licenses changes over time, and their specific values depend on the source data used. The following representative table is from Black Duck. This table reports the popularity of licenses in the OSS projects that Black Duck tracks as of August 2014 (only licenses with at least 5% popularity are shown; more detail can be found at http://www.blackducksoftware.com/resources/data/top-20-open-source-licenses):

Table 1. Popularity of OSS Licenses

Rank	License	%
1	GNU GPL 2.0	26%
2	MIT license	18%
3	Apache license 2.0	15%
4	GPL 3.0	11%
5	BSD 3-clause	7%
6	Artistic License (Perl)	5%
7	GNU LGPL 2.0	5%

Source: Black Duck

A variant of the permissive approach is to release software without any copyright restrictions, but this can involve some surprising complexities. Information without copyright restrictions is sometimes said to be in the *public domain* in the copyright sense.[4] In the United States, works by U.S. Federal Government employees that are created as part of their official duties are not normally subject to copyright within the United States.[5] There have been some debates on whether or not it is even possible to truly disclaim copyright in United States in other cases, but for this paper, all that matters is the final legal effect. A common mechanism for achieving similar results (when the author is not a U.S. Federal Government employee) is to use the Creative Commons CC0 license, a.k.a. the Universal (CC0 1.0) Public Domain Dedication, which dedicates the work to the public domain (in the copyright sense). However, the CC0 and similar licenses were not specifically designed with software in mind; in particular, it is unclear

[4] The term "public domain" means something different when it is used in the context of export control regulations; the term instead means "information that is published and generally accessible to the public," including via sales and conferences. Thus, "copyright public domain" and "export control public domain" are different; most people mean copyright public domain when they say "public domain."

[5] The U.S. Government can receive copyrights, including copyrights from contractors, and it can also assert copyright in other countries.

whether they can effectively disclaim warranties as part of the license. In addition, these dedications can create barriers; many organizations (such as the Apache Software Foundation) are unprepared to deal with software that has no copyright claim.

All of the above apply to the general case when releasing a new OSS project. When releasing changes to an existing OSS project, it is typically necessary to at *least* release the software under the OSS license used by the project (since it will usually not be accepted otherwise). If the new OSS project must tightly integrate with other projects, those other projects' licenses must be considered, and in some cases it may be wise to adopt them.

We suggest that researchers use the MIT license if they have no idea of what license to apply. This permissive license is extremely short and simple (http://opensource.org/licenses/MIT), and it is easy to transition to other licenses in later versions if desired. The BSD 2-clause license is practically equivalent to the MIT license and is also a reasonable choice. This guide suggests the MIT license as the default because it is simple but explicitly lists allowed actions; GitHub also recommends it for those who "want it simple and permissive." However, all of the licenses listed above are popular, and any of them can be a reasonable choice depending on other factors.

The license text for software should, at least, be included in a file at the top of its directory with a conventional file name (e.g., the filename LICENSE or COPYING, possibly with the .txt or .md extension).

It is best if every file also include near the top a copyright notice followed by either the text of the license or brief text that identifies the license. A good copyright notice is: "Copyright [year project started] - [current year], [project founder] and the [project name] contributors." Copyright notices are not strictly required legally, but they are easy to add and provide some modest advantages [Balter2015]. This could be followed by text that states the license, at least "Licensed under the <license name including its version number>." The license name could be the full name, or at least the official SPDX name of the license. This helps later developers and reviewers ensure that the software is legally licensed.

A significant amount of software is released without any license, but researchers should avoid this mistake. In most cases software without a license statement means that the default copyright laws apply, and in most cases the defaults are that nobody else may reproduce, distribute, or create derivative works without permission. Software without a license statement is not usually OSS, because others cannot legally use it in most circumstances. Courts may find in some cases an implied right, but developers should work to stay out of courts instead of requiring a court to decide what rights were intended (the court may make a determination very different from what was intended). Before 1976, software had to include a copyright statement to be copyrighted, but that was a

long time ago, and essentially all countries (including the United States) impose copyright restrictions by default on any work immediately on creation in a fixed form (e.g., as text).

Documentation can be released under OSS licenses, but if the focus is on creating documentation, it might be best to choose a license designed for documentation while using an OSS approach. In general, researchers should choose a license for releasing *Free Cultural Works* (as defined in https://creativecommons.org/freeworks). This includes material licensed under the CC-BY, BY-SA, or CC0 licenses from the Creative Commons. In these cases, list those licenses instead or in addition to the OSS licenses. If researchers have no idea what license to choose, we suggest using CC-BY, for similar reasons.

F. Contributor agreements and assignments

OSS projects need to set up a legal mechanism to enable legal contributions, enable whatever it wants to do later, and reduce the risk of later legal problems. By definition, all OSS projects select OSS license(s) that are included as part of the work, as described above, to set up a basic legal framework.

There are additional legal mechanisms that some OSS projects use. Two of the most common are contributor agreements and contributor (copyright) assignments. Contributor agreements can, in turn, be divided into a Developer Certificate of Origin (DCO) and other contributor agreements.

The additional legal mechanisms of contributor agreements and contributor assignments can have legal benefits and might reduce certain legal risks, but many of them also carry risks to the success of the entire project. Contributor agreements other than DCOs, and especially contributor (copyright) assignments, can significantly reduce the number of potential contributors, potentially leading to failure of the entire OSS project. This means that contributor agreements beyond DCOs and contributor assignments include a significant risk of project failure that attorneys do not always mention, so researchers and PMs need to consider all risks (not just the legal risks that attorneys consider). This section describes the contributor agreement and contributor assignment approaches, their benefits, and their drawbacks, as summarized in Figure 4.

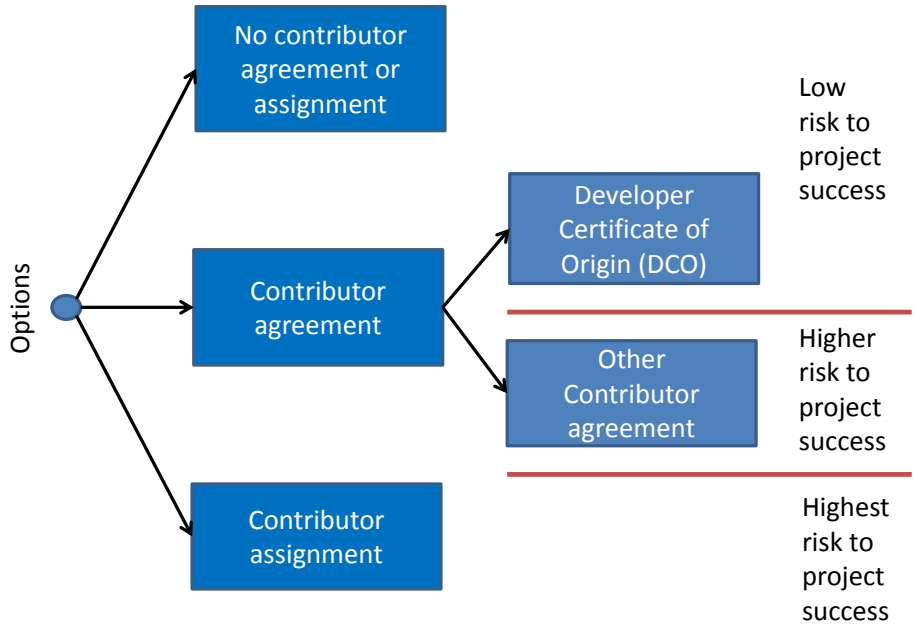

Figure 4. Options for Contributor Agreements and Contributor Assignments

A contributor agreement, also termed a contributor license agreement (CLA), makes an agreement or assertion when contributing change(s). A contributor agreement may be physically signed, digitally signed, or even an unsigned electronic assertion. A contributor agreement typically includes the following (see [Corbet/Bottomley2014]):

1. The assertion that the contributor has the right to contribute the code
2. An identification of the contribution itself
3. An agreement (consent) that the code can be distributed under the project license.

An especially simple type of contributor agreement is a Developer Certificate of Origin (DCO) (http://developercertificate.org/). In this approach the contributor agrees, as part of the contribution, to provide an electronic certificate of origin that the contribution meets the contributor agreement points listed above. These are asserted directly by the contributor, instead of requiring the contributor to get someone else to sign them (as noted in their name, a developer certificate of origin is provided by the *developer*). In addition, these are typically not physically signed, and are instead electronically referenced when submitting a contribution. Their usual implementation is usually a one-line note that the contributor agrees to the DCO terms. Since they are directly provided by the contributor (instead of being from the contributor's employer), are normally not physically signed, and are easy to add, DCOs are easy to do and have little risk of inhibiting contributions. Projects that use DCOs include the Linux kernel [Linux2011], git, and Docker [Docker2014].

Some projects choose to require other kinds of contributor agreements, including those signed by an employer (if one exists) and/or physically signed contributor agreements. A benefit of these agreements is that they can reduce risk if someone later asserts that a contribution was not legal. A signed agreement by someone authorized by the employer provides a strong legal defense if an employer later claims that an employee's contribution was not legally provided. A physically signed contributor agreement is considered by some to be stronger evidence that the agreement was valid. These other contributor agreements are not as difficult to get as copyright assignments (to be discussed), but they can still have a significantly detrimental effect on contributions because obtaining them can be time-consuming and can either cause the contributor to not contribute (even if they legally could) or can cause a significant delay. If a signature must be from someone authorized to speak for a company (typically an employer), this can impose significant delays on a contribution, and if an employer is too busy this mechanism can inhibit perfectly legal contributions. Physical signatures require mailing and tracking, creating an extra processing burden. Thus, other kinds of contributor agreements inhibit or delay some contributions and can increase the risk that the OSS project will not receive enough contributions to continue. What's more, it is not clear that agreements actually provide significantly more legal protection; organizations generally cannot verify that the signer in another organization is authorized to do so, or is even the person they claim to be, and generally cannot verify a physical signature before the fact. Thus, the legal protection assignments these approaches are supposed to provide may prove elusive. Others who have looked into the issues also argue against using contributor agreements beyond DCOs in most cases [Kuhn2014].

An assignment transfers legal rights, usually the copyright, to another party. Thus, they are also called copyright assignment agreements (CAA). A potential benefit of an assignment is that it can provide some additional legal defense against someone who claims, for example, that they hold the copyright to some portion and that the OSS project cannot distribute that portion. In particular, in the United States, only a copyright holder has standing to sue for copyright infringement, so an organization that wants to enforce a license in court on copyright grounds needs copyright in at least some portion of the software. Of course, the assignment might not be valid (perhaps the person who signed the assignment was not authorized to do so), but the risk is somewhat reduced because many people will be willing to sign an assignment only if they believe they have the authority to do so. Assignments can make it easy to release the software under a different license; without an assignment it may be impractical to release the software under a different license. Some business models may also require assignment (e.g., if the OSS is released under a protective license and a company intends to sell proprietary versions of it).

However, assignments significantly increase the risk of OSS project failure. Many organizations will not sign assignments even if they *will* sign contributor agreements. Even if they will sign assignments, they might only sign them under unusual circumstances. Also, getting assignments signed often inserts a long delay that takes longer than merely getting an agreement. Such assignments significantly inhibit and slow contributions because (1) they must often be specially signed by contributors' company lawyers or management and (2) they create risks for the contributors because they lose rights that they would otherwise be able to execute. Such assignments are especially risky for contributors if the assignment is a for-profit entity (instead of a non-profit entity). The incentives of the for-profit entity might be very different than those of the contributors, and the contributors would typically have no recourse if the assignee decides to take a direction the contributors do not desire. Another problem is that most assignments assume there is a copyright to grant; in the United States, a work developed by a U.S. Government employee as part of his or her duties does not have copyright protection in the United States, making an "assignment" a complicated legal question. In short, assignments can risk project success because of the time it takes to get such agreements (slowing progress) and because many potential contributors cannot or will not sign these assignments. In addition, organizations generally cannot verify that the signer of another organization is actually authorized to sign, making their legal strength weaker than might first appear.

None of these mechanisms are legally *required* to create an OSS project; most OSS projects do not have contributor agreements or contributor assignments. Submitting a contribution with the project license has the effect of releasing the software under that license. Laws already forbid contributing information that a contributor has no right to provide (including the laws governing copyright and fraud). What is more, doing nothing is easy. In short, the primary "competition" for agreement and assignment approaches is simply to not use them at all.

Some successful OSS projects do require a DCO. The Linux kernel, for example, has been ongoing for more than two decades without any copyright agreements (it instead uses a DCO). What is more, the Linux kernel has overall fared quite well in court cases claiming copyright infringement, in spite of rigorous legal attack.

We recommend that researchers normally have a DCO or no agreement requirement at all. In particular, we recommend that projects do not require copyright assignments unless they have a compelling need for one. Anything that significantly impedes contributions also risks viability of the whole project and should only be considered with caution. However, alternatives may be appropriate if the business model requires it or there are unusually high legal risks (e.g., that companies will later retract their employees' contributions).

G. Discuss decisions publicly using the OSS project's collaborative environment

Once the OSS project's collaborative environment is established (such as its website, discussion system, version control system, and trackers), *use* that environment to publicly discuss and record all decisions – even if all the current participants are within a single organization. This way, other participants can see the decisions that were previously made and why (e.g., so they won't repeat past mistakes), and potential contributors are more likely to trust that you will treat their contributions fairly (since the decisions are being made publicly).

There are various ways of doing this. On GitHub, comments specific to a particular proposal (via an issue or pull request) might be discussed as responses to the issue or pull request, while broader issues might be discussed more broadly (e.g., via a mailing list).

In a few cases it may be necessary to discuss issues more privately, in which case, do so, but try to limit them.

7. OSS Project Inputs and Results

A project, when viewed as a system, receives a number of inputs (in particular contributions) and produces results (in particular software, documentation, and related information such as its website). These are, of course, deeply related: inputs that are accepted become part of the project results. Some practices that are good software development practices in general become more important in OSS because they can enable collaboration. Below, we discuss key points about OSS projects inputs and outputs, with an emphasis on those that are especially important when developing an OSS project to enable technology transition.

A. Workflows and requirements for accepting technical contributions

We previously discussed the reuse of existing components (the reuse of unmodified components was discussed in section A, and the reuse of modified ones were discussed in section B). We also previously discussed the role of standards (in section C).

We now turn to a fundamental task in any project: encouraging, processing, and accepting technical contributions. There are many kinds of technical contributions; they include bug reports, feature requests, proposed changes to the code, and documentation improvements. The workflow (process) for submitting technical contributions, and the minimum requirements for them, should be documented so that submitters will know what is required. We will cover each in turn.

1. Workflow for technical contributions

There are many different possible workflows, aka processes, for accepting technical contributions. When there is only a single developer, an extremely simple workflow is possible. However, even then, it is useful to have tools that track potential changes to allow comparisons and reversions to past versions. As the number of developers increases, it becomes even more important to use tools and processes to efficiently handle contributions.

Various technical terms have emerged to describe actions and workflows:

- A *repository* is simply a database that records the results of a project, including current and past versions, along with a record of who changed what and when.
- A *centralized version control system* (the older method of doing version control) supports a single central repository that developers work with.

- A *distributed version control system* (the newer method of doing version control, implemented by tools such as git) enables a peer-to-peer approach to version control; each user has at least one complete repository, and the version control system supports communication between interacting repositories. In general we recommend using a distributed version control system today, in particular git.
- A *pull request* occurs "when a developer asks for changes committed to an external repository to be considered for inclusion in a project's main repository." [Johnson2013].
- The *upstream* and *downstream* of a repository or project refer to that repository or project's intended location in a supply chain. The terminology is an analogy to a river; those who receive a product or service are downstream, while those who produce products and services are upstream. A project is often in the middle, and thus is downstream from some and upstream of others. Projects that make changes to the products they use should typically send changes back upstream so that everyone (including themselves) will receive those changes merged with other useful changes.

Distributed version control systems enable many workflows that are difficult to do with the older centralized version control systems, and thus some workflows may be unfamiliar to some developers. The key is to pick one that works (for now) and document it; workflows can be changed later. Here are some examples based on *Pro Git* and other sources:

1. *Fully centralized workflow*: A centralized repository is used, and all authorized developers are authorized to directly update its contents. This workflow is common with centralized version control systems and with developers who are primarily familiar with them. However, this approach is limiting, and those limitations can slow development.
2. *Feature branch workflow/GitHub flow*. This uses a centralized repository that all authorized developers can modify, as with the fully centralized workflow, but all feature development takes place in a dedicated branch for each feature instead of the master branch. A master branch is used to identify the current version of the software. When the new feature is ready to be merged into the master branch, a developer can merge the branch into the master branch or send a pull request for the feature to be merged into the master branch. This requires developers to create new branches for new features, but it is far more flexible than the fully centralized workflow.

   The *GitHub flow* workflow, advocated by GitHub, adds one additional rule: Anything in the master branch must be deployable (see https://guides.github.co

m/introduction/flow/). Scott Chacon argues that the GitHub flow approach is useful because of its simplicity; see http://scottchacon.com/2011/08/31/github-flow.html. GitHub recommends that you *not* wait for a feature to be complete before submitting a pull request; these early requests can enable additional review and feedback (see https://github.com/blog/1124-how-we-use-pull-requests-to-build-github).

3. *Gitflow workflow (aka git-flow).* This approach, advocated by Vincent Driessen, uses more branches, e.g., "master" for the full releases and "develop" for the latest merged version in development, with feature branches starting from the "develop" branch. Once a feature is completed it is merged into "develop," and after some point that combination is merged into "master." Often this workflow term implies that there is a centralized repository that all authorized developers can modify. One advantage of the gitflow workflow is that it clearly separates the current development version from the release versions, which can be an advantage if there is a separate review process that takes a long time and cannot be put into a continuous integration process. However, it is more complicated than many other workflows, leading to the potential for some confusion.
4. *Integration-Manager Workflow*. Each developer has write access to their own public repository and read access to everyone else's public repository. Typically there is a *blessed* repository that can be read by all, but only directly changed by the integration manager. Developers modify their private repositories and push to their individual public repositories, and then send a pull request to the integration manager(s) when they want their changes integrated into the blessed version.
5. *Dictator and Lieutenants Workflow*. This is a variant of a multiple-repository workflow. It's typically used only when there are hundreds of simultaneous collaborators (e.g., the Linux kernel). This workflow is essentially a recursive application of the integration-manager workflow. In this workflow, various integration managers (called lieutenants) are in charge of certain parts of the repository, and the lieutenants have one integration manager known as the benevolent dictator. The benevolent dictator's repository serves as the reference repository from which all the collaborators need to pull their copies.

Some sources for more information on workflows include the *Pro Git* section on distributed workflows at http://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows and the Atlassian tutorial at https://www.atlassian.com/git/tutorials/comparing-workflows/centralized-workflow.

In addition, when using a tool like git, anyone who is not an authorized developer can still propose a change. The external developer can fork an existing repository

(typically some blessed or central repository), modify it (possibly by first making a private copy and then pushing its contents to a public one), and then submit a pull request to someone authorized to accept it.

The best workflow is, in the end, whatever workflow is acceptable and works for the project participants. That said, we suggest that new projects seriously consider applying the GitHub flow workflow to their authorized developers, as it is relatively simple yet flexible. The GitFlow workflow should also be considered, especially if there is a separate special process for transitioning development software into released software (beyond its continuous integration process). The integration-manager workflow should be considered if the software is highly sensitive yet has contributors from many potentially less-trusted developers, since it only allows one or a few developers to directly modify the blessed repository. Note, however, that workflow processes can be changed much more easily than other decisions such as licenses; it is quite common to start with a simple workflow, and increase its formality as the project grows in size.

2. Requirements for technical contributions

A key issue with technical contributions is that the project should encourage their submission, yet insist on a reasonable quality.

1. For bug reports, there needs to be enough information so that the bug can be fixed by developers. The *most important item* in a bug report for developers is the set of steps necessary to reproduce the defect, according to one survey of OSS project developers [Bettenburg2008]. In short, developers must be able to duplicate a problem before they can find and fix it. Thus, bug reporters should be encouraged to provide enough information so that the incorrect behavior can be reproduced and identified as such. This also suggests that programs should be designed so that behaviors are easier to reproduce.
2. For feature requests, there needs to be enough information to understand the suggestion.
3. For code changes, the code should provide the expected functionality with adequate quality. Here review is often critical. Initial contributors will not know all of the project's local conventions, so it is unsurprising when a reviewer replies with specific issues to fix. However, reviewers need to provide enough feedback to the contributor so that the contribution can be updated and re-submitted.
4. For documentation, the information must be clear, accurate, and useful to its intended audience.

Projects should not accept proposed changes (particularly code changes) of poor quality, but if a project waits for idealized perfection it will wait forever. A project

should work to clearly identify its minimum expectations, identify with specificity the problems with a proposed change that keeps it from being accepted, and focus on incremental improvement.

Work to ensure that *existing* material meets your quality expectations (and if not, specifically document why it doesn't and ask for help in that area). Contributors will look to existing material as examples of what you expect, and in any case, it's hypocritical to ask people to "do as I say, not as I do."

Many projects use Wikis, where anyone in the group (or possibly anyone) can make immediate changes to at least some of the documentation and/or the website but those changes can be trivially reverted. In these cases, projects make a conscious decision to at least temporarily accept proposed changes because their impact tends to be smaller, but they can still reject or modify the changes later if they are inadequate.

It is important to provide timely and specific feedback to proposed changes. Timeliness is especially important for new contributors; new contributors will often start by proposing a small change, to see if the project is live and open to contributions. New contributors will also typically not know enough to be able to make sweeping changes. If contributors receive timely feedback on their proposed changes, and it is clear what is necessary for the changes to become acceptable, the contributors are more likely to be able to make the changes.

The long-term goal should be to *encourage* quality contributions from a variety of people and organizations, since this leads to sustained activity in the OSS project.

The following sections discuss potential requirements on inputs and results to ensure that they are quality contributions and results.

B. Code and test contribution requirements

Code and test contributions should meet whatever coding style guideline the project uses. Coding style guidelines pre-resolve questions such as how the code should be indented, how names should be used, and what code constructs should be used or avoided. A consistent coding style makes the code easier to understand, avoids common mistakes, and also makes the software easier to integrate. For example, if someone reformats code in the process of making a change, it can be harder to re-integrate that change into a larger work. Coding style guidelines also reduce many kinds of arguments, and help people focus instead on technical content.

On new projects, code style guidelines are selected by whoever starts the project, and are often established by pointing to some existing guideline(s). Coding style guidelines must cover issues for the specific programming language(s) to be used and

should generally use prevailing practices for those languages (where they exist). Examples include:

- GNU Coding Standards http://www.gnu.org/prep/standards/standards.html
- Linux kernel Coding style https://www.kernel.org/doc/Documentation/CodingStyle
- Google style guides for OSS projects https://github.com/google/styleguide, including the Google Java style guide http://google.github.io/styleguide/javaguide.html
- Mozilla Coding Style: https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Coding_Style.

C. Change (commit) formats and rewriting history

Proposed changes should focus on a specific logical change, since this enables code review, and be individually short enough to review. A single change that rewrites most of the program is typically impractical to review. Since most OSS developers use git, we will discuss git here, although the principles are the same for other version control tools (especially distributed version control tools).

In practice, the need for logical changes means that git users should create at least one separate git branch for each different logical set of changes they wish to make (e.g., there might be a branch for adding a new type of input file, and a different branch for improving the performance of some internal calculation algorithm). These different branches make it easier to keep logically different changes separate. Inside a branch there may be one or more commits that lead up to the desired logical change (e.g., the first one may refactor the system to make it easier to change, followed by a logical progression of commits that lead to the desired change in total).

Sometimes development by a single developer does not proceed linearly. For example, changes may start to go in one way, the developer may realize that does not work, and then other changes go in a different way to obtain a final result. The developer may constantly commit even minor changes to ensure their progress (or what they hope is progress) is not lost. The developer may also constantly update their local copy from the upstream project, in which case the history of their changes will be interspersed with changes from the upstream project. When this happens, there is an important question: should the developer submit all that history (including directions that did not pan out and merges from upstream) or should the developer submit a cleaner version that shows what a logical progression *would* have looked like (had it been followed)?

There are many reasons to *not* include every irrelevant twist and turn in a set of local commits. It is difficult to review these complex changes, since many of those

changes will be later changed. Also, tools like "git bisect" (which can help find where a bug was introduced) are harder to use if some commits partially work or introduce known problems that will be fixed in later stages.

Thus, projects may ask committers to provide a simplified commit history to simplify review, especially once the project has made significant progress. Git provides tools to help do this, including git rebase and git cherry-pick. With these tools a developer can rewrite a local repository's history to make development look like a simple logical sequence of steps – even when reality was more complicated. Developers may often rewrite *private* history (history that has not been sent elsewhere) so that the final commitment is a simple, logical progression of changes that can be easily reviewed.

However, this desire for a simpler history can lead newcomers to a serious mistake: rewriting *public* history in an already public branch. It is fine to rewrite a branch's history in a local repository as long as it has not been sent out publicly. For example, if a developer likes to commit changes every few minutes using "git commit," without publishing those changes elsewhere, then using rewriting commands that rewrite commands (such as "git commit --amend") can be perfectly fine.

However, *never* rewrite public history. Once a branch's information is copied out to others' repositories (including being made public), it will have a specific sequence of cryptographic values that identify each commit (including the final one). Any attempt to change a public branch will produce different cryptographic values; this can be confusing and possibly look like an attack. Instead, simply create *another* (different) branch with the rewritten history, and ask people to pull from that rewritten branch instead.

More information is available in the "Pro Git" section on rewriting history[6] and the Atlassian git tutorial on rewriting history.[7]

D. Use pre-packaged components

Sometimes proposed inputs will add dependencies on other components. In general, encourage the consideration of prepackaged components for reuse, but evaluate these components before they are used to ensure that they are acceptable (see section A). Where practical, use existing package management systems to simplify acquiring and updating these other reused components (typically through the network).

Most operating systems today include a package management system for managing application software installed on them and for downloading additional or updated

---

[6] https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History

[7] https://www.atlassian.com/git/tutorials/rewriting-history/

packages, and in practice there may be more than one package manager on a given system. On some systems, such as traditional Linux distributions and the *BSDs, these package management systems are also used to manage software libraries. Systems such as Red Hat Enterprise Linux, CentOS, and Fedora store operating system packages in the rpm format and can download packages using tools such as yum or dnf. Systems such as Debian and Ubuntu store operating system packages in the deb format and can download packages using tools such as apt-get. Windows installer is the native method for installing applications on Microsoft Windows. On Apple MacOS systems the Mac App Store is the official digital distribution platform, while Homebrew is a popular package manager based on git. Mobile systems (such as Android and iOS) typically have at least one app store available to them for downloading and installing end-user applications.

Many programming language ecosystems today include a system for quickly getting and managing libraries for that language from some common repository. These systems include a centralized package repository with corresponding package conventions (at least a standard data format), along with tools for automatically acquiring and updating software from the centralized repository. Examples of languages with a common repository, and a common repository used by that language, are shown in Table 2.

Table 2. Examples of Common Language-specific Repositories

Language	Repository
Clojure	Clojars (http://clojars.org/)
Common Lisp	QuickLisp (https://www.quicklisp.org/)
Go	GoDoc (http://godoc.org)
Haskell	Hackage (http://hackage.haskell.org/)
Java	(Maven) Central Repository (http://central.sonatype.org/ ; see also the search system at http://search.maven.org/
Javascript	npm (http://npmjs.org) especially for server-side and Bower (http://bower.io/) especially for client-side
OCaml	OCaml Package Manager aka OPAM (http://opam.ocaml.org/)
Perl	Comprehensive Perl Archive Network (CPAN) (http://www.cpan.org/)
PHP	Packagist (https://packagist.org/)
Python	Python Package Index (PyPI) (https://pypi.python.org/pypi)
Ruby	RubyGems.org (https://rubygems.org/)
Rust	Crates (https://crates.io/)

It may also be convenient to build on containers. Docker containers are extremely popular today. The Open Container Project is an effort backed by the Linux Foundation to unite various projects, including Docker, to create a standard format for containers.

As of 2015, many repositories, especially those for programming languages, are not cryptographically signed [VersionEye2014]. Some repositories do not even have basic protections against man-in-the-middle attacks.[8] Thus, before using a package management system, check that it has at least basic protection against man-in-the-middle attacks (such as using https for all uploads and downloads). If it does not, the software that is used may not be the same as the software originally developed; mitigations such as cross-checking with the project website or getting the package manager to improve may be needed. The good news is that users are increasingly demanding that repositories improve their security, and specifications and libraries (such as "The Update Framework" at http://theupdateframework.com/) are being developed and implemented to enable repositories to easily meet a minimum bar for security.

ModuleCounts maintains a list of some package management systems for individual languages and the number of components in them (http://www.modulecounts.com/). Wikipedia maintains a list of software package management systems at https://en.wikipedia.org/wiki/List_of_software_package_management_systems.

E. Use current common build tools and conventions.

Developers of OSS may build (compile) software on a variety of platforms, for a variety of platforms. Thus, while it is a good to have a flexible build system for any system, it is especially important when developing OSS, because this enables more developers and users. It is generally best to prefer common OSS build tools and conventions so that potential developers can concentrate on improving the software instead of trying to understand an unusual build system. Note that some tools identified here as "build tools" do much more than older tools called build tools, e.g., many also support downloading and configure dependencies, testing, and/or configuration and installation.

When writing software in Java, common build tools include Gradle, Maven, Ant, and Apache Buildr. When writing software in C/C++, common build tools include the autotools (especially autoconf and automake) and cmake; directly using "make" (particularly GNU make) is also relatively common. Developers who use make directly may want to consider using ":=" or the new-to-POSIX "::=" assignment because these assignment statements eliminate the exponential growth in time that can occur when using make's traditional "=" assignment.

[8] The HOST project identified the lack of man-in-the-middle protection on the Cygwin project in 2015. Cygwin is a project that provides many OSS packages to Microsoft Windows users. The Cygwin project agreed and fixed the problem (in this case by switching to using only https).

When compiling to object files on Unix-like systems, a common and important convention to support is the DESTDIR convention. This is a simple convention where if the make macro "DESTDIR" is set, that value is prepended to any filename when files are copied on installation. This convention is a necessary precondition for many packaging systems, and thus following this convention can make software easier to install.

One mistake is assuming that your development or build system always has network connectivity, then checking or downloading software. Platforms that execute final build systems are often intentionally disconnected from the Internet (to make them hard to attack), falsifying this assumption. Most programs that implement build systems support caches that make it possible to safely preload software ahead of time; if yours does not, add any necessary mechanisms or simply create a subdirectory for each reused component for use as a cache for the purpose.

Another mistake is using a *recursive make* approach. As software gets large, a common structuring method is to use directories and subdirectories to divide it up. By itself, this is fine. However, a common mistake is to attempt to recursively call each subdirectory to have each subdirectory built separately, even when the subdirectories are interrelated components of a larger system (and not totally isolated separate projects). When this is done, the build system never has complete information on what has been changed and requires rebuilding; the result is often incorrect results and/or lengthy build times. Build systems should have the complete information for software so that they have the correct information to perform a build. It is fine to separately build independent components, and the *data* for subdirectory builds can be divided into subdirectories; the key is that the build system should be provided with the correct information. It may be counterintuitive, but a non-recursive make is often significantly faster than a recursive make. The recursive make approach can happen in any build system, whether or not it uses make. For more information on the problems of recursive make, see [Miller1998].

The instructions for building should be included with, and maintained, just like the rest of the source code. For more on common build tools and conventions (including DESTDIR), see [Wheeler2013c].

F. Regression test suites and continuous integration

Include in the software an easily invoked regression test suite. Use common conventions, e.g., if your build system uses makefiles it should support invoking the test suite via "make check" (similar conventions exist in other build systems). In this way, developers can easily do a local check that their changes do not break existing functionality, and they are more likely to add tests if the regression test suite is already present.

A regression test suite may exist but fail to be thorough. One way to measure thoroughness is by measuring the branch coverage of the regression test suite. Some OSS projects, such as SQLite, achieve 100% branch coverage in their regression tests. These regression tests should include unit tests (testing at a unit/component level) and functional tests (which test the functionality of the software as a whole). Few projects today achieve 100% branch coverage, but even moderately thorough regression tests can immediately detect many defects in a proposed change, and in particular make code refactoring much more effective.

In addition, use continuous integration. Historically, regression tests were re-run every night if at least one change occurred after rebuilding the software (the *daily build*). However, the current best practice (due to increased availability of computing power) is to re-run the build and regression test suite after every push into selected branches; this is called a *continuous build process* or *continuous integration*. Continuous integration is often implemented using a continuous integration server, which monitors every change to the source code; each change triggers automatic analysis, rebuilding, and regression testing (including unit testing and extensive functional testing). This monitoring can be triggered or even implemented by version control mechanisms such as git's hooks. Continuous integration is the logical follow-on to daily (or nightly) builds because it speeds identification of problems even faster than daily/nightly builds.

Continuous integration makes it easier to quickly identify *exactly* what change caused a problem so that it can be fixed. Examples of continuous integration servers that help implement continuous integration include CruiseControl, Jenkins, and Tinderbox.

G. Develop secure software

Of course, software needs to work securely, countering the attacks that are inevitable in today's environment. In particular:

1. *Strive to develop secure software throughout its development*. Be aware of, and counter, common types of vulnerabilities. Ensure that the program checks every input, particularly from untrusted users, against a whitelist to ensure that the inputs meet expected formats (e.g., are syntactically correct and within expected ranges); reject all other inputs. When using an SQL database, use prepared statements or similar mechanisms that counter vulnerabilities. Design the program so that it has the least necessary privileges. In general, develop the software in a way that reduces the likelihood of security vulnerabilities or their impact. Such approaches also tend to make software more robust and reduce debugging time.
2. *Use tools to look for unknown vulnerabilities so that they can be detected and repaired*. The DHS Software Assurance Marketplace (SWAMP) at

https://continuousassurance.org/ enables software to be uploaded and analyzed by a variety of tools. *State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation* [Wheeler2014b] identifies many types of tools and techniques for analyzing software and helps identify the various technical objectives that they can help achieve.

3. *Use tools to look for known vulnerabilities in included or required software.* Check that any included or required software (e.g., libraries) does not have known vulnerabilities, and in particular, update those libraries to versions without known vulnerabilities. This requires periodic re-examination because vulnerabilities may have been found since the library was selected earlier. Origin analysis tools can help identify known vulnerable components; OWASP DependencyCheck is one such tool, and Sonatype, Black Duck, Codenomicon, and VersionEye have a variety of such products.
4. *Include both positive and negative tests in the regression test suite.* Many regression test suites only include correct inputs (positive tests) as test cases. For security, this is a mistake. The regression test suite should also include negative tests, that is, inputs that should be rejected. For numbers, include numbers that are too small, too large, or incompatible with other values. For text, include the null string, text that is not in the current locale (typically these are byte sequences that are not UTF-8 or UTF-16), overly large text strings (if that applies), and text that does not meet the parsing requirements.

H. Software releases

Software undergoes continuous improvement; it is important to indicate to users that a particular version is "done" and ready for use.

Released software should have a simple and clear version number that clearly indicates to potential users that this is *current* compared to previous versions. One of the most common versioning conventions today is the "semantic versioning" system described at http://semver.org/ which uses the version number format MAJOR.MINOR.PATCH, where you increment:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backward-compatible manner, and
- PATCH version when you make backward-compatible bug fixes.

Major version zero (0.y.z) is reserved for initial development. In initial development anything may change at any time and the public API should not be considered stable.

The semantic versioning system is easily distinguished from decimal numbers because it uses two periods (not just one). Some versioning systems use only one period, but then it is not clear whether 3.2 is before or after 3.11; in contrast, programs for comparing version numbers can unambiguously determine that 3.2.0 is always before 3.11.0 (since this is the semantic versioning convention). Another advantage of semantic versioning is that users can estimate, at a glance, the difficulty of upgrading to the new version. Still another advantage of semantic versioning is that it can easily support multiple development lines, e.g., 3.0.7 and 2.8.6 might both be current releases, and users can immediately understand their differences (there must have been an incompatible API change). Some use release dates as a version number; if they are done in ISO date order (YYYYMMDD) then it is easy to see what is first, but this scheme does not easily handle having more than one actively supported branch (a common situation).

Avoid creative version numbering schemes. For example, Frama-C uses the element names corresponding to an atomic number (e.g., "Sodium" for major version 11), while TeX adds an extra digit to its version number to asymptotically approach pi (as of June 2015 its version number is 3.14159265). In many cases people must assemble thousands or hundreds of thousands of programs (including libraries transitively). Managing large number of programs requires automated tools to manage dependencies and upgrades, and idiosyncratic version number systems interfere with these tools. They also make it difficult for users to determine whether they are up to date, e.g., is Sodium before or after Neon?[9] Codenames or informal names can be fine, but only if they are closely tied to clear version numbering schemes like semantic versioning.

Releases should be provided to potential users in ways that they will find easiest to use. If the software is a library, try to get the library included in a common library system for that programming language, using a common format for libraries in that language. Similarly, if it is an application for a common operating system, it may be wise to package it for that operating system. See section D, which discusses packaging systems, but now consider it in the context of *releasing* software in these forms.

Be sure to include, with each release, brief information for users on *why* the new release should (or should not) be installed. In particular, users will want to know whether it fixes security vulnerabilities (and if it does, which ones), and whether it will be easy to update to (e.g., if it changes an API, including any storage formats).

---

[9] Sodium is after Neon, because Sodium's atomic number is 11 and Neon's is 10. Note that determining which is first is not trivial for most people. In contrast, determining whether "11.0.0" is before or after "10.0.0" is extremely easy.

I. Continuously improve

It is not possible to do everything at once, nor is it necessary. Instead, focus on continuously improving. One reason to prefer common conventions is that others, who are expert in specific tools or processes, can more easily help you apply related tools to your specific circumstance.

8. Tips for Making an OSS Project Successful

Many actions can increase – or decrease – the likelihood of OSS project success. This section contains a collection of tips for making OSS projects successful.

A key source for the first two subsections is the work by Charles M. Schweik and Robert C. English of the Massachusetts Institute of Technology (MIT). They performed 5 years of painstaking quantitative research to answer the question, "What factors lead some open source software (OSS) commons (i.e., projects) to success, and others to abandonment?" [Schweik2012]. Their work divided OSS projects into two phases, initiation (before the first public software release) and growth (after the first public software release), and identified the key issues for each phase. Quotes in the next two subsections are from [Schweik2012] unless noted otherwise.

A. Tips for initiation

During initiation (before the first public software release), Schweik and English determined that the following are the most important issues (in order of importance) for OSS project success (the "you" here applies to the researcher):

1. "Put in the hours. Work hard toward creating your first release." The details in chapter 11 tell the story: If the leader put in more than 1.5 hours per week (on average), the project was successful 73% of the time; if the leader did not, the project was abandoned 65% of the time. They are not saying that leaders should put in only 2 hours a week; instead, the point is that the leader must consistently put in time for the project to get to its first release.
2. "Practice leadership by administering your project well, and thinking through and articulating your vision as well as goals for the project. Demonstrate your leadership through hard work…."
3. "Establish a high-quality web site to showcase and promote your project."
4. "Create good documentation for your (potential) user and developer community."
5. "Advertise and market your project, and communicate your plans and goals with the hope of getting help from others."
6. "Realize that successful projects are found in both [GPL and non-GPL licenses]."

7. "Consider, at the project's outset, creating software that has the potential to be useful to a substantial number of users." Remarkably, the minimum number of users is surprisingly small; they estimate that successful growth stage projects typically have at least 200 users. In general, the more potential users, the better.

B. Tips for growth

Similarly, once OSS projects switch to growth these are the factors that matter in order of importance (again, "you" is the researcher):

1. "Your goal should be to create a virtuous circle where others help to improve the software, thereby attracting more users and other developers, which in turn leads to more improvements in the software…." Researchers should do this the same way it is done in initiation: spending time, maintaining goals and plans, communicating the plans, and maintaining a high-quality project web site. The user community should be actively interacting with the researcher's development team. Researchers should make it easy for people to get to try out the software. Similarly, researchers should make it easy for potential collaborators to improve the software and submit those improvements.
2. "Advertise and market your project." In particular, successful growth projects are frequently projects that have added at least one new developer in the growth stage.
3. Have some small tasks available for contributors with limited time.
4. Welcome competition (e.g., other OSS projects that do similar things), as "competition seems to favor success." Competition often encourages projects to do better.
5. Consider accepting offers of financing or paid developers (they can greatly increase success rates). This point, in particular, should surprise no one. Researchers should not automatically reject financial offers, as long as they are legal, contribute to the overall goal, and avoid ethical problems such as conflicts of interest. PMs should be prepared to discuss potential collaboration and try to find ways to manage issues such as potential conflicts of interest.
6. "Keep institutions (rules and project governance) as lean and informal as possible, but do not be afraid to move toward more formalization if it appears necessary."

There is strong empirical proof that "adding more developers [during the growth stage] causes success" [Schweik2012,170], so researchers should have a strong focus on adding more developers (at least one) from a different organization once the project's software has been initially released. There are many reasons for this. Obviously

additional developers can speed development in general, and their different skill sets will often make previously difficult problems easier. Additional developers also reduce the perception of risk by potential users and developers; should the original developer stop working on the project, the project is more likely to continue because there are other developers. The development team itself can be small; many successful OSS projects have only two to three primary developers. However, projects with more developers can directly tackle larger problems.

The majority of successful growth projects have greater than 1,000 downloads and at least 200 users [Schweik2012, 272-273]. This, unsurprisingly, suggests that OSS projects should strive to be general enough to support at least that many potential users. A user may eventually become a collaborative developer or hire one; if this happens often, the project itself will become very active. Projects cannot be "all things to all people," but it is important to provide enough capability to enough people so collaboration will continue.

None of this is surprising, but it is confirmed by quantitative data analysis. Some goals, such as keeping complexity low, were not differentiators for success. Indeed, successful projects tended to have a little more complexity than abandoned projects. This does not mean a project should strive for complexity. Instead, both successful and abandoned projects often strive to reduce complexity—so complexity is not something that distinguishes them. Also, a project that focuses on user needs probably is more complex than one that does not, simply because user needs can sometimes be complex.

C. Other issues, including those to do with standard conventions

Other tips for establishing and running an OSS project can be found in [Scott2011], [Fogel2009], [Raymond2000], and [Gabriel2005].

In general, researchers should try to identify and follow standard conventions, prefer the use of widely used tools, and apply widely accepted best practices (some of these have been noted above). Doing so makes it easier for potential collaborators to begin collaborating. For example, researchers should strongly consider the following:

1. *Use git for version control*. As noted above, git is a widely used OSS distributed version control tool, and its use simplifies collaboration with other projects. Git has a reputation for being hard to use, but more recent versions and services have made it easier to use. Other tools that might be useful include mercurial and subversion; however, mercurial and bazaar are less common, and subversion cannot support distributed version control (it only supports centralized version control).
2. *Use widely used coding style conventions for the selected programming language(s).* For more, see section B.

3. *Prefer the use of current common build tools and conventions*. For more information, see section D.
4. *Include an easily invoked regression test suite and use continuous integration.* For more information, see section F.
5. *Ensure the software is secure*. For more information, see section G.

9. Overcoming Impediments

Simply developing and releasing software can be useful (e.g., for documenting past research so it can be repeated), but it does not necessarily create a successful OSS project. The preceding guidelines should increase the probability of success. However, there are some potential impediments to OSS projects that are common and have specific corrections. Here are a few common impediments and ways to counter them.

A. Users/developers do not know of the project

A common problem is that users and potential co-developers are unaware of the OSS project. A clear (preferably unique) project name can help them find the project, once they know of it, but a unique name will not (by itself) help them discover that it exists in the first place.

It is critical to have a front-page website that clearly describes the purpose of the project; this can help those who are searching for it. This will help those specifically seeking projects like it, especially once others start linking to the project front page (since this increases search rankings). However, many users will not know to seek out the project, and obscure projects often have low rankings in web searches.

OSS project leaders will typically want to identify those who might be especially interested in the project, and make them aware of it. This can include contacting individuals who are interested in the area, mailing lists, groups, and organizations, Briefings at related conferences can also help spread the word. At the least, identify web sites that have pages that are especially relevant, and ask them to add a link to the project front page (providing them specific text that is appropriate to insert in their specific page). The project leader may also want to try to get interviewed on various relevant podcasts (e.g., FLOSS Weekly) and blog sites (e.g., Slashdot), or have articles written about the project (e.g., in lwn.net or opensource.com). In short, an OSS project needs to "market" its existence, especially at first, so that those who might be interested in the project can learn about it.

B. Missing functionality

All software could have additional functionality added, and many users will want functionality that the current software lacks. There are steps that can make missing functionality less of an impediment in some cases; they can be grouped into those making

it easy to work around missing functionality and those making it easy to add that functionality.

There are many ways to design software so that users can more easily work around missing functionality. This includes using standard data formats (e.g., JSON, XML, and various specific specifications), providing a programming API (e.g., a REST API) so other programs can control your software, and storing data in databases that can be separately queried. This way, users can send and extract data through your program to other systems that can add missing functionality.

There are also many ways to design software so that users can more easily add functionality. This includes providing a plug-in architecture, clean design and code, and thorough current developer documentation. Good names (e.g., for classes and methods) can make a program much easier to modify. Following common practices and including a regression test suite also makes it easier to add functionality. These apply to any software, but for OSS these can be especially important. Some potential collaborators start by wanting to make only a small change; by making small changes easy to do, more substantial changes are more likely to be contributed as well.

C. Inadequate quality or trustworthiness

Users are far less likely to use buggy software, especially if those defects directly affect their primary reasons for using the software. OSS projects should use a variety of techniques to provide and maintain good quality. This includes general quality issues (e.g., functions that perform incorrectly), but also security issues (e.g., enabling attack). This is even true for security-related programs; a security-related program can itself have vulnerabilities that *enable* instead of counter attack. Since no single mechanism guarantees high quality, a variety of approaches should be used; some of them have already been noted above.

Compiler warning flags should be maximally enabled to detect potentially erroneous constructs. The code should be written to try to compile warning-free (no warnings are reported even when flags are maximally enabled). This is an ideal; some warning flags cannot be practically enabled for some projects, and some warnings may be unavoidable. However, avoiding dangerous constructs where practical can reduce defects, including those that lead to vulnerabilities.

Refer to some coding guide, so that instead of arguing about what formatting or construct is okay, the project can focus on the actual functionality (see section B). Refactor the software while changing it so it will be easier to maintain in the future. There should be general striving to make the software code clear, since complex code is more likely to have hidden defects.

Static analysis tools should be used to detect code quality issues, especially code vulnerabilities. As noted above, source code can be sent to the SWAMP for analysis for potential vulnerabilities. Many proprietary tool vendors have projects for evaluating OSS to find vulnerabilities ahead of time.

Regression tests should be thorough and included with the project, and combined with continuous integration (see F). The software should be developed to be secure in general (see G).

D. Low trust

Users may not trust a project, even if the code itself is trustworthy. Thus, it is important to make it clear to users why they should trust the project.

Don't just perform the tasks in the previous section, but make it clear to users what is being done to provide high quality, with evidence to back it up. To the extent possible, make decisions transparently and with a good rationale; this provides evidence that future decisions will be reasoned and not capricious. Limit the promises made, but strive to keep them.

People are often more willing to trust a project if many others trust it. Thus, focusing on improving the software so that it will help many users – and growing that user base – can also help gain the trust of others.

E. Excessive user cost/time/effort

Potential users have limited time, and co-developers in OSS projects often start as users. Thus, any barrier for user actions can lead to dramatically fewer users and developers. Consider tasks from the user's point of view and constantly work to reduce their cost, time, and effort. This includes:

- Reduce the cost/time/effort to initially try out the software. Users will typically want to briefly try out the software. Provide ways to try out the software as quickly as possible. This could be by providing a setup that runs just their web browser, by providing a prebuilt container (e.g., as a Docker container) or virtual machine, or by providing simple installation mechanisms that just work on common platforms. Using standard installation systems (e.g., cmake or autotools for compiled programs) and working with packagers to ensure the software is easy to package and widely available as a package also helps. Simple introductory tutorials (video, audio, and/or documentation) can be helpful as well.
- Reduce the cost/time/effort to learn/train. Users who decide that the software may be valuable or have decided to start using it (typically on a small pilot

project) will need to learn how to use the software. Ideally the software's user interface would be so obvious that no training is needed; this ideal is sometimes impractical, but where possible, build on knowledge and interfaces that your users are likely to already know. Provide documentation and/or more detailed tutorials to help people learn.

- Reduce the cost/time/effort to use/apply. Software that is easy to learn but hard to use practically is not very welcome. If the program has a user interface, it is often instructive to simply give a program to a potential user and ask them to perform a task while explaining what they are doing and why. This can reveal confusing aspects of user interfaces and missing functionality needed in the real world. It can also reveal where a user interface is simply too clumsy to use repetitively. If the program is a library, implement a standard API and/or have an API that is easy to use for simple common cases. Wisely chosen defaults and simple interfaces for common cases can help provide deep functionality when necessary while still leaving the software easy to use for most cases.
- Reduce the cost/time/effort to integrate/onboard the software. Provide standard interfaces, and in particular accept or export data formats that are in common use for that kind of software. It should be possible to incrementally increase the use of the software; few people will want to "bet the company" on software they've never seen before.

Efforts to reduce cost, time, and effort must typically continue over time. The good news is that once a project begins gaining contributions (because it has reduced that effort to some reasonable level), other contributions can help reduce the effort further.

10. Conclusions

Researchers (including principal investigators) and program managers (PM) can use an OSS approach to successfully transition technology into successfully deployed solutions. OSS approaches can build on existing work, either by modifying existing components and/or using those components to build new systems.

New projects require establishing a collaborative environment, including selection of hosting services, a governance process, and license, along with the issues involving contributor agreements and contributor assignments. A variety of common conventions for OSS project inputs and results can improve the likelihood of success, as can tips derived from previous projects. Common impediments should be identified and overcome.

Any technology transition process requires planning, time, and effort. Full technology transfer using an OSS approach requires submitting changes to existing OSS projects and/or creating a self-sustaining project. This is not difficult, and many people have done it, but it still requires planning, time, and effort. It also need not happen all at once. But technology transition is vital – technology research typically only helps people if it is transitioned to those who need it. OSS approaches can be a valuable way to practically implement technology transition.

References

URLs are subject to change without notice.

[Balter] Balter, Ben. "Copyright notices for open source projects." *Ben.balter.com*. June 3, 2015. http://ben.balter.com/2015/06/03/copyright-notices-for-websites-and-open-source-projects/

[Bettenburg2008] Bettenburg, Nicolas, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. "What Makes a Good Bug Report?" *Proceedings of ACM SIGSOFT 2008/FSE-16*. November 9–15, 2008. Atlanta, Georgia. http://thomas-zimmermann.com/publications/files/bettenburg-fse-2008.pdf

[Biere2004] Biere, Armin. *The Evolution from LIMMAT to NANOSAT*. Technical Report #444. Dept. Computer Science, ETH Zurich, CH-8092 Zurich, Switzerland. April 15, 2004.

[Carlson2006] Carlson, Curtis and William Wilmot. *Innovation – The Five Disciplines for Creating What Customers Want*. August 8, 2006. Published by Crown Business. ISBN 978-0307336699.

[Corbet/Bottomley2014] Corbet, Jonathan and James Bottomley. "The most powerful contributor agreement." *LWN.NET*. April 2, 2014. https://lwn.net/Articles/592503/

[D'Amico2013] D'Amico, Anita, Brianne O'Brien, and Mark Larkin (Secure Decisions). "Building a Bridge across the Transition Chasm." *IEEE Security & Privacy*. March/April 2013. http://securedecisions.com/building-a-bridge-across-the-transition-chasm/

[DHS-Mobile2014] *Broad Agency Announcement Solicitation HSHQDC-14-R-B0015 Project: Mobile Technology Security*. https://www.fbo.gov/utils/view?id=c12401cd9aec08ddbc64c5495d4bb2ad

[Dion-Schwarz2010] Dion-Schwarz, Cynthia (Director, Information Systems & Cyber Security, Office of the Director, Defense Research and Engineering). "DoD Current State for Software Technology Readiness Assessments." *Systems & Software Technology Conference*. April 2010. http://ieee-stc.org/proceedings/2010/pdfs/CD2694.pdf

[Docker 2014] Stinemates, Nick. "Introducing the Docker Developer Certificate of Origin." *Docker blog*. January 7, 2014. https://blog.docker.com/2014/01/docker-code-contributions-require-developer-certificate-of-origin/

[DoD2009] Department of Defense (DoD) Chief Information Officer (CIO). *Clarifying Guidance Regarding Open Source Software (OSS)*. October 16, 2009. http://dodcio.defense.gov/Portals/0/Documents/FOSS/2009OSS.pdf

[DoDFAQ] Department of Defense (DoD) Chief Information Officer (CIO). DoD *Open Source Software (OSS) FAQ – Frequently Asked Questions regarding Open Source Software (OSS) and the Department of Defense (DoD).* http://dodcio.defense.gov/OpenSourceSoftwareFAQ.aspx

[Fogel2009] Fogel, Karl. Producing Open Source Software: How to Run a Successful Free Software Project. 2009.

[Gabriel2005] Gabriel, Richard P., and Ron Goldman. Innovation Happens Elsewhere: Open Source as Business Strategy. Morgan Kaufmann. April 2005. http://dreamsongs.com/IHE/

[Github2014] GitHub. "Open source licensing." GitHub Help. Retrieved 2014-07-25. https://help.github.com/articles/open-source-licensing

[Jelliffe] Jelliffe, Rick. "Where to get ISO Standards on the Internet free." http://archive.oreilly.com/pub/post/where_to_get_iso_standards_on.html

[Johnson2013] Johnson, Mark. "What is a pull request?" http://oss-watch.ac.uk/resources/pullrequest

[Kegel2004] Kegel, Dan. Contributing to Open Source Projects HOWTO. http://www.kegel.com/academy/opensource.html

[Kuhn2014] Kuhn, Bradley. "Why Your Project Doesn't Need a Contributor Licensing Agreement." June 9, 2014. http://ebb.org/bkuhn/blog/2014/06/09/do-not-need-cla.html

[Linux2011] Developer Certificate Of Origin (DCO). http://elinux.org/Developer_Certificate_Of_Origin

[Maughan2013] Maughan, Douglas, David Balenson, Ulf Lindqvist, and Zachary Tudor. "Crossing the 'Valley of Death': Transitioning Cybersecurity Research into Practice." *IEEE Security & Privacy*. 2013. Volume 11, Number 2, pp. 14-23. http://www.csl.sri.com/papers/ieee-sp-tt-2013/

[Miller1998] Miller, Peter A. Recursive Make Considered Harmful, AUUGN Journal of AUUG Inc., 19(1), pp. 14-25. 1998. http://aegis.sourceforge.net/auug97.pdf

[Morin2012] Morin, Andrew, Jennifer Urban, and Piotr Sliz. "A Quick Guide to Software Licensing for the Scientist-Programmer". *PLOS Biology*. July 26, 2012. http://www.ploscompbiol.org/article/info:doi/10.1371/journal.pcbi.1002598

[NRC2004] National Research Council (NRC) of the National Academies. *Accelerating Technology Transition: Bridging the Valley of Death for Materials and Processes in Defense Systems*. National Academies Press. ISBN 0-309-09317-1 (Book) and 0-309-54583-8 (PDF)

[Raymond2000] Raymond, Eric S. *Software Release Practice HOWTO*. http://en.tldp.org/HOWTO/Software-Release-Practice-HOWTO/

[Rogers2003] Rogers, Everett M. *Diffusion of Innovation*. 2003. 5th edition. New York, N.Y.: Free Press. As cited in [NRC2004] page 8.

[Schindlholzer2008] Schindlholzer, Bernhard. "Developing Better Value Propositions Using the NABC Framework". *www.diametrics.io*. November 18, 2008. http://www.diametrics.io/developing-better-value-propositions-using-the-nabc-framework.html

[Schweik2012] Schweik, Charles M. and Robert C. English. *Internet Success A Study of Open-Source Software Commons*. 2012. ISBN 97800-262-01725-1.

[Scott2011] Scott, John, David A. Wheeler, Mark Lucas, and J.C. Herz. *Open Technology Development (OTD): Lessons Learned & Best Practices for Military Software*. May 16, 2011. http://dodcio.defense.gov/Portals/0/Documents/FOSS/OTD-lessons-learned-military-signed.pdf

[Spolsky2002] Spolsky, Joel. *Strategy Letter V*. June 12, 2002. http://www.joelonsoftware.com/articles/StrategyLetterV.html

[SRI2011] SRI International Best Practice. http://blog.twg.ca/wp-content/uploads/2011/10/SRI-NABC-Approach.pdf

[VersionEye2014] VersionEye. Which programming language has the best package manager? 2014. http://blog.versioneye.com/2014/01/15/which-programming-language-has-the-best-package-manager/

[vonHippel2005] von Hippel, Eric (Professor of Management of Innovation and Head of the Innovation and Entrepreneurship Group at the MIT Sloan School of Management). *Democratizing Innovation*. 2005. Cambridge, MA: MIT Press. ISBN 0-262-00274-4. http://evhippel.mit.edu/books/

[Walker2014] Walker, Molly Bernhart. "DHS launches Mobile Technology Security project." *FierceMobileGovernment*. June 18, 2014. http://www.fiercemobilegovernment.com/story/dhs-launches-mobile-technology-security-project/2014-06-18

[Weir2010] Weir, Rob. "The Recipe for Open Standards (and Why ISO Can't Cook)". *Robweir.com*. 2010-09-09. http://www.robweir.com/blog/2010/09/recipe-for-open-standards.html

[Wheeler2011a] Wheeler, David A. "Publicly Releasing Open Source Software Developed for the U.S. Government." *Journal of Cyber Security & Information Systems (CSIAC) (formerly Software Tech News)*. Vol: 14 Num: 1. February 2011. https://www.csiac.org/journal_article/publicly-releasing-open-source-software-developed-us-government

[Wheeler2011b] Wheeler, David A. *How to Evaluate Open Source Software / Free Software (OSS/FS) Programs*. Revised as of August 5, 2011. http://www.dwheeler.com/oss_fs_eval.html

[Wheeler2013a] Wheeler, David A. *What is open security?* Institute for Defense Analyses (IDA). August 21, 2013. IDA Document D-4993.

https://www.ida.org/~/media/Corporate/Files/Publications/IDA_Documents/ITSD/2014/D-4993.pdf and http://www.dwheeler.com/essays/open-security-definition.html

[Wheeler2013b] Wheeler, David A., and Tom Dunn. *Open Source Software in Government: Challenges and Opportunities*. August 2013.

[Wheeler2013c] Wheeler, David A. *Releasing Free/Libre/Open Source Software (FLOSS) for Source Installation*. 2013. http://www.dwheeler.com/essays/releasing-floss-software.html

[Wheeler2014a] Wheeler, David A. *Why Open Source Software / Free Software (OSS/FS, FLOSS, or FOSS)? Look at the Numbers!* May 8, 2014. http://www.dwheeler.com/oss_fs_why.html

[Wheeler2014b] Wheeler, David A. and Rama S. Moorthy. *State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation*. Institute for Defense Analyses Paper P-5061. July 2014. http://www.acq.osd.mil/se/initiatives/init_pp-sse.html

Acronyms and Abbreviations

Below is a glossary of key abbreviations used in this guide. Well-known abbreviations are omitted (such as "U.S." for United States). This list does not include terms that are not abbreviations (e.g., "LLVM" is not an acronym, it is the full name of the project).

AFL	Academic Free License *or* American Fuzzy Lop
API	Application Programming Interface
ASF	Apache Software Foundation
ATP	Automated Theorem Proving
BSD	Berkeley Software Distribution
CAA	Copyright Assignment Agreement
CLA	Contributor License Agreement
CPL	Common Public License
CSD	Cyber Security Division
CSS	Cascading Style Sheets
DCO	Developer Certificate of Origin
DHS	Department of Homeland Security
DoD	Department of Defense
FAQ	Frequently Asked Questions
FSF	Free Software Foundation
GNU	GNU's Not Unix
GPL	General Public License (GNU's)
HOST	Homeland Open Security Technology
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force (IETF)
IRCA	Identify candidates, Read existing reviews, Compare the leading programs' basic attributes to the needs, Analyze
ISO	International Organization for Standardization (sic)
MIT	Massachusetts Institute of Technology
MPL	Mozilla Public License
NDA	Non-Disclosure Agreement
NIST	National Institute of Standards and Technology
OSI	Open Source Initiative
OSL	Open Software License
OSS	Open Source Software
OWASP	Open Web Application Security Project
OSR	Open Standards Requirement
PM	Program Manager

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE (DD-MM-YY)	2. REPORT TYPE	3. DATES COVERED (From – To)
24-11-15	Final	

4. TITLE AND SUBTITLE

Using an Open Source Software Approach for Cybersecurity Technology Transition

5a. CONTRACT NUMBER

N66001-11-C-0001, subcontract D6384-S5

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBERS

6. AUTHOR(S)

David A. Wheeler

5d. PROJECT NUMBER

GT-5-3329

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES

Institute for Defense Analyses
4850 Mark Center Drive
Alexandria, VA 22311-1882

8. PERFORMING ORGANIZATION REPORT NUMBER

P-5279
H 15-000816

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Joshua L. Davis
Georgia Tech Research Institute
250 14th Street NW, Room 256, Atlanta, GA 30318

Funding provided by:
Department of Homeland Security

10. SPONSOR'S / MONITOR'S ACRONYM

GTRI & DHS

11. SPONSOR'S / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

Project Leader: David A. Wheeler

14. ABSTRACT

This document provides guidance on how to use an open source software (OSS) approach to support technology transition. Using an OSS approach can enable and speed technology transition, but many researchers and government program managers are unfamiliar with how to use it. The goal of this guide is to help researchers (including principal investigators) turn their ideas into successfully deployed solutions, and also to help program managers as they select research proposals and oversee researchers' work. It discusses how to build on existing work (including how to evaluate existing software and modify existing OSS), establish a collaborative environment for a new OSS project when necessary, the key issues in OSS project inputs and results, tips for making an OSS project successful, and approaches for overcoming common impediments.

15. SUBJECT TERMS

open source software, technology transition, cybersecurity, research, cybersecurity research, computer security, software development, principle investigators, researchers, program managers, software evaluation, collaborative development, collaborative environment

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Joshua L. Davis
Unclassified	Unclassified	Unclassified	Unlimited	78	19b. TELEPHONE NUMBER (Include Area Code) 678-831-0182