UCI-AM-13-115

ACQUISITION RESEARCH PROGRAM
SPONSORED REPORT SERIES

# Streamlining the Process of Acquiring Secure Open Architecture Software Systems

08 October 2013

**Dr. Walt Scacchi, Senior Research Scientist**

**Dr. Thomas A. Alspaugh, Project Scientist**

Institute for Software Research

**University of California, Irvine**

## About the Authors

**Walt Scacchi—**Scacchi is a senior research scientist and research faculty member at the Institute for Software Research, University of California, Irvine. He received a PhD in information and computer science from UC Irvine in 1981. From 1981 to 1998, he was on the faculty at the University of Southern California. In 1999, he joined the Institute for Software Research at UC Irvine. He has published more than 150 research papers and has directed 60 externally funded research projects. In 2011, he served as co-chair for the 33rd International Conference on Software Engineering—Practice Track, and in 2012, he served as general co-chair of the 8th IFIP International Conference on Open Source Systems.

Dr. Walt Scacchi, Senior Research Scientist
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
E-mail: wscacchi@ics.uci.edu

**Thomas Alspaugh—**Alspaugh is a project scientist at the Institute for Software Research, University of California, Irvine. His research interests are in software engineering, requirements, and licensing. Before completing his PhD, he worked as a software developer, team lead, and manager in industry, and as a computer scientist at the Naval Research Laboratory on the Software Cost Reduction or A-7 project.

Dr. Thomas Alspaugh, Project Scientist
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
E-mail: thomas.alspaugh@acm.org

THIS PAGE INTENTIONALLY LEFT BLANK

UCI-AM-13-115

ACQUISITION RESEARCH PROGRAM
SPONSORED REPORT SERIES

# Streamlining the Process of Acquiring Secure Open Architecture Software Systems

08 October 2013

**Dr. Walt Scacchi, Senior Research Scientist**

**Dr. Thomas A. Alspaugh, Project Scientist**

Institute for Software Research

**University of California, Irvine**

THIS PAGE INTENTIONALLY LEFT BLANK

## Table of Contents

# Executive Summary

The goal of this research was continuing to investigate a new approach to address challenges in the acquisition of secure open architecture (OA) software systems for the Department of Defense (DoD; Department of Defense Open Systems Architecture [DoDOSA], 2011). Program managers, acquisition officers, and contract managers will increasingly be called on to review and approve security measures employed during the design, implementation, and deployment of OA systems. Our efforts sought to make this a simpler, more transparent, and more tractable process. Such a process must be easy to reuse, adapt, and streamline for different system application domains in order to realize cost reductions and improve acquisition workforce capabilities.

Our research described in this report focuses on two problems for acquisition research: (1) how best to acquire secure OA software systems that include reusable software product line components (Mactall & Spruill, 2012; Womble, Schmidt, Arendt, & Fain, 2011); and (2) how to articulate and streamline a process for identifying and reviewing the security of OA software systems. Our overall research results presented in the following chapter stipulate that the best ways to streamline the process for acquiring secure OA systems in line with DoD's Better Buying Power 2.0 ("What is Better Buying Power?" 2013) guidelines are those that: (a) encourage the adoption of open (source) business models; (b) provide open source models of acquisition processes; and (c) employ techniques for streamlining acquisition processes for secure OA systems through direct measurement and assessment of acquisition processes, redesign and evolution of acquisition processes, design of new acquisition processes specific to secure OA systems, and through employment of cost management as an element in the design of future OA system acquisition processes.

Our prior research demonstrated how complex OA systems can be designed, built, and deployed with alternative components and connectors into functionally similar system versions, to realize for overall system security. The case study research methods we continue to employ highlight opportunities for cost reduction through transparent system security requirements specification and OA system acquisition process streamlining. We believe our results can apply to both mission-critical software systems within the DoD, as well as to enterprise software systems in other government agencies and industrial firms. For example, in the second and fourth sections of this report, we present the results that focus on the case studies centering on military command and control systems, such as the future C2RPC models being considered by naval commands (Garcia, 2010; Gizzi, 2011), in line with the multi-party engineering agile adaptive ecosystem (MPE/AAE) envisioned for

the Defense Information Systems Agency and other government agencies (Defense Information Systems Agency [DISA], 2012; Reed, Benito, Collens, & Stein, 2012).

Finally, our principal research results are documented in five research publications that are included in this report in the following chapters. These follow from our research vision for this 2012–2013 effort that is summarized in the research overview presented in the first chapter. Then we present our overall research results for streamlining the process of acquiring secure open architecture software systems. Next, we explore the technical challenges with the development of processes in securing open architecture software systems. Then, we elaborate the results from the second chapter in the context of challenges in the development and evolution of secure open architecture command and control systems, where next-generation military command and control systems are expected to be developed from reusable software (product line) components that must realize a secure OA system, even though individual components may not necessarily be secure. Then we seek to generalize some of the challenges and results focusing on component-based OA software systems in different application domains, including those of interest to the DoD and other government agencies, where available software component capabilities that come with "provisionments" (i.e., we must determine which of our problems we face can be solved with available components) are seen to displace a traditional focus on software system "requirements" (we determine what capabilities we need in order to solve the problems we face). This, we believe, may represent a fundamental shift of how future acquisition processes and practices may be performed, especially as we transition to development of OA software systems that are composed using commercially available software components, such that our functional requirements must be addressed through the provided component capabilities or provisionments. Finally, we explore possible ways for systematically specifying the security requirements (or provisionments) of secure OA systems composed from software components whose security may not be known in advance.

## References

What is Better Buying Power? (2013). Retrieved from http://bbp.dau.mil/

Defense Information Systems Agency (DISA). (2012). *DoD open source and community source software development in Forge.mil* (SoftwareForge Document ID–doc26066doc26066). Retrieved from http://www.disa.mil/News/Conferences-and-Events/DISA- Mission-Partner-Conference-2012/~/media/Files/DISA/News/Conference/2012/DoD_Open_Source_Community_Forge.pdf

Department of Defense Open Systems Architecture (DoDOSA). (2011, December). *Contract guidebook for program managers*, Vol. 0.1. Retrieved from https://acc.dau.mil/OSAGuidebook

Garcia, P. (2010). Maritime C2 strategy: An innovative approach to system transformation, in *Proceedings of the 15th International Command & Control Research & Technology Symposium*, Paper 147, Santa Monica, CA.

Gizzi, N. (2011). Command and control rapid prototyping continuum (C2RPC) transition: Bridging the valley of death, in *Proceedings of the 8th Annual Acquisition Research Symposium*, Vol. 1, Naval Postgraduate School, Monterey, CA.

Guertin, N., & Clements, P. (2010). Comparing acquisition strategies: Open architecture versus product lines. In *Proceedings of the Seventh Acquisition Research Symposium* (Vol. 1, pp. 78–90). Naval Postgraduate School, Monterey, CA.

Guertin, N., & Womble, B. (2012). Competition and the DoD marketplace. In *Proceedings of the Ninth Acquisition Research Symposium* (Vol. 1, pp. 76–82). Naval Postgraduate School, Monterey, CA.

Reed, H., Benito, P., Collens, J., & Stein, F. (2012). Supporting agile C2 with an agile and adaptive IT ecosystem. In *17th International Command and Control Research and Technology Symposium* (ICCRTS), Paper-044. Fairfax, VA.

Mactal, R., & Spruill, N. (2012). A framework for reuse in the DoN. In *Proceedings of the Ninth Acquisition Research Symposium* (Vol. 1, pp. 149–164). Naval Postgraduate School, Monterey, CA.

Womble, B., Schmidt, W., Arendt, M., & Fain, T. (2011). Delivering savings with open architecture and product lines. In *Proceedings of the Eighth Acquisition Research Symposium* (Vol. 1). Naval Postgraduate School, Monterey, CA.

THIS PAGE INTENTIONALLY LEFT BLANK

# Streamlining the Process of Acquiring Secure Open Architecture Software Systems

## Research Overview

### Introduction

This research focuses on continuing investigation and refinement of techniques for streamlining the process, reducing the costs, and improving the readiness of future workforce for the acquisition of complex software systems. Emphasis was directed at identifying acquisition process streamlining and cost reduction opportunities within secure open architecture (OA) systems combining best-of-breed software components and software product lines (SPLs) that are subject to different security requirements.

This chapter provides an overview of the research effort during the period of September 10, 2012, through September 9, 2013. It includes a statement of work and description of the four research activities engaged during this period, followed by identification of the two acquisition research problems being investigated, the research and development basis for our research, and identification of our research publications that contain our studies and results. Each section is presented in turn.

### Statement of Work and Research Description

In our studies at hand, we investigate secure open architecture (OA) systems that incorporate SPL techniques that include proprietary and open source software (OSS) components, and overall configurations are subject to different security requirements. The combination of SPLs and OSS components within secure open architecture systems represents a most significant opportunity for reducing the acquisition costs of software-intensive systems by the DoD and other government agencies. This effort builds on both our prior acquisition research and related acquisition research efforts at the PEO IWS (Guertin & Clements, 2010; Guertin & Womble, 2012; Womble, Schmidt, Arendt, & Fain, 2011), Department of the Navy (Mactal & Spruill, 2012), and Software Engineering Institute (SEI; Bergey & Jones, 2010; Jones & Bergey, 2011) that address SPLs, as well as SEI efforts addressing OSS (Hissam, Weinstock, & Bass, 2010).

OSS is recognized as an integrated web of people, processes, and organizations, including project teams operating as virtual organizations (Scacchi, 2007, 2009, 2010). There is a basic need to understand how to identify an optimal mix of OSS within OA systems as products, production processes, practices, community activities, and multi-project (or multi-organization) software ecosystem. However, the relationship among OA, OSS, security requirements, and acquisition is poorly understood (cf. Scacchi, 2009, 2010; Scacchi & Alspaugh, 2011, 2012b; Naegle & Petross, 2007). Subsequently, in 2007–2008, we began by examining how different OSS licenses can encumber software systems with OA, which therefore give rise to new requirements for how best to acquire software-intensive systems with OA and OSS elements (Scacchi & Alspaugh, 2008). As a result of our most recent prior efforts (Alspaugh, Scacchi, & Asuncion, 2010; Scacchi & Alspaugh, 2012b), we have been able to demonstrate that for enterprise information systems, which are widespread throughout the DoD and the U.S. government, it is both possible and feasible to develop systems that incorporate best-of-breed software components, whether proprietary or OSS, in ways that can reduce the initial and sustaining acquisition costs of such systems. Doing so, however, requires new guidance and ideally, automated tools, for explicitly modeling and analyzing the architecture of an OA system during its development and evolution, along with modeling the annotating the architecture with software component license rights and obligations. Our results thus demonstrate a significant technological advance in the acquisition and development of OA systems, as a breakthrough in negotiating and simplifying software license analyses throughout the contracting activities. Creating similar advances for secure OA systems is the next breakthrough needed, and the results of our efforts to create such advances are presented in this report.

Overall, our research project sought to articulate acquisition research problems with respect to the issues that determine what types or kinds of answers can be realized through this investigation. Subsequently, we proposed a straightforward approach to the proposed effort that focused on four activities:

- Investigating the interactions between software system acquisition guidelines and processes, software system requirements, requirements for OSS components, and consequences of alternative software system architectures that incorporate different mixes of OSS components, SPLs that combine both OSS and proprietary software components with open APIs and open standards (Scacchi & Alspaugh, 2008, 2012b). This entailed exploring the balance between development, verification, and validation of property and security rights, as well as contractual obligations within continuously improving OSS system elements while managing the evolution of OA systems at design-time, build-time, and release and run-time. The main results of our studies in this activity area appear in our research publications elsewhere in this report (Scacchi & Alspaugh, 2013a, 2013b, 2013c, 2013d).

- Developing formal foundations for establishing acquisition guidelines for use by program managers seeking to provide software-intensive systems in cost-reducing ways that rely on development and deployment of secure OA systems using OSS and SPL technology and processes (Alspaugh, Asuncion, & Scacchi, 2009b; Alspaugh, Scacchi, & Asuncion, 2010; Scacchi & Alspaugh, 2011, 2012b). The main results of our studies in this activity area appear in our research publications elsewhere in this report (Alspaugh & Scacchi, 2013a, 2013b).
- Continuing to develop concepts for the design of an automated system that can support acquisition of secure OA systems so as to (a) determine their conformance to acquisition guidelines/policies, contracts, and related license management issues, and (b) to facilitate and support future acquisition workforce with the skills to properly review, approve, and manage the acquisition of complex systems that incorporate a secure OA (Alspaugh, Scacchi, & Asuncion, 2010; Scacchi & Alspaugh, 2011, 2012b). The main results of our studies in this activity area appear in our research publications elsewhere in this report (Scacchi & Alspaugh, 2013b).
- Documenting the investigation, foundations, and results of the research in (a) a technical report delivered within 30 days of project completion to the technical point of contact at NPS (i.e., this report submitted by October 9, 2013); (b) a research paper (Scacchi & Alspaugh, 2013a) included in the online *Proceedings of the 10th Annual Acquisition Research Conference*, in Monterey, CA, May 2013 (see the next chapter of this report); and (c) related research venues and publications (see the last five chapters of this report).

To help motivate the approach and results developed through this study, we provide some background on emerging issues in the acquisition of software-intensive systems that require open architectures and encourage or embrace the utilization of OSS, such as rapid, distributed evolution to meet immediate warfighter needs and its interplay with validation and system management. Next, we describe the body of the proposed research effort. This covers the problem, issues, opportunities, and approach for acquisition research we identify. Finally, we present the primary form of our research results, documented in six research papers appearing as the last five chapters of this final report for 2012–2013.

## Acquisition Research Problems

Two lines of inquiry follow from our accomplishments described above and in detail elsewhere (Scacchi & Alspaugh 2011, 2012). One is how our results might shed light on secure software systems whose OAs conform to a *reusable software product line*, while the other is how our approach might be extended to also address the semantic modeling and analysis of *a process for specifying and reviewing secure OA system requirements.*

### ***Research Problem: Acquiring Secure OA Systems That Include Reusable SPL Components***

Organizing and developing software product lines (SPLs) relies on the development, use, and reuse of explicit software architectures (Bosch, 2001; Clements & Northrop, 2003). However, the architecture of an SPL is not necessarily an OA—there is no requirement for it to be so. Thus, we are interested in discussing what happens when reusable SPLs may conform to an OA, which is a concern we share with others (Guertin & Clements, 2010; Mactal & Spruill, 2012). In particular, our interest here is to consider how to cost effectively achieve a secure OA system that incorporates SPL components that are subject to different intellectual property licenses (Alspaugh, Asuncion, & Scacchi 2009a, 2009b; Alspaugh, Scacchi, & Asuncion, 2010) and to different security requirements (Scacchi & Alspaugh, 2011, 2012).

Three considerations come to mind. First, if the SPL is subject to a single homogeneous security policy and intellectual property license for each system element confined to its own security containment vessel (Scacchi & Alspaugh, 2011), then the complexity of the security scheme may act to reinforce a vendor lock-in situation and potentially increase system costs. Motivating factors for OA include the desire to avoid such lock-in, and to embrace open innovation and open market competition (Guertin & Womble, 2012), whether or not the SPL components have open or standards-compliant APIs.

Second, if an OA system employs a secure reference architecture much like we have in the design-time architecture depicted in Figure 1, which is then instantiated into different software product line configurations, as suggested in the secure build-time architecture shown in Figure 2, then such a design-time architecture effectively defines a reusable SPL consisting of possible different system instantiations composed from similar components instances (e.g., different but functionally similar Web browsers, word processors, email, calendaring applications, relational database management systems). However, alternative build-time or run-time versions can be accommodated with the same or different security requirements. This flexibility merits further study that can reveal the elements and criteria for a streamlined acquisition process, where alternative OA system configurations may be easier or lower cost to realize, yet still satisfy security requirements. Identifying such a process and how to tailor and streamline for different types of OA software systems (e.g., enterprise information systems, command and control systems, weapon systems) that can be developed with reusable SPL system components, is thus the focus of our proposed research effort.

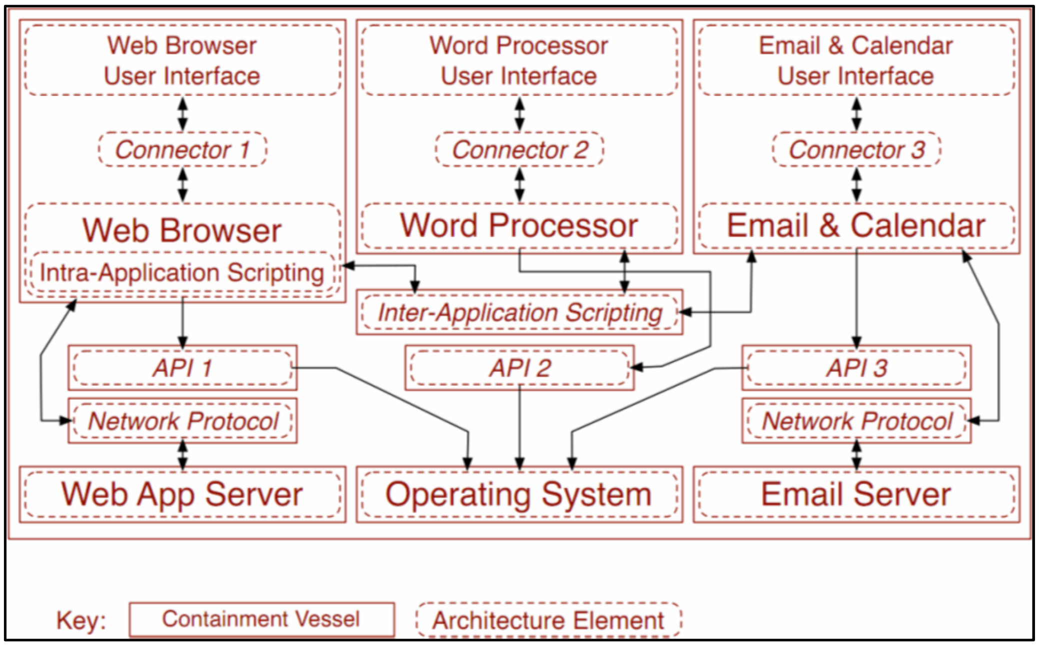


**Figure 1. A Design-time Open Architecture for a Secure Enterprise Information System Software Product Line Consisting of a Web Browser, Word Processor, Email, and Calendaring Applications, Hosted on a Network of Servers and Host Operating System**
(Scacchi & Alspaugh, 2012)

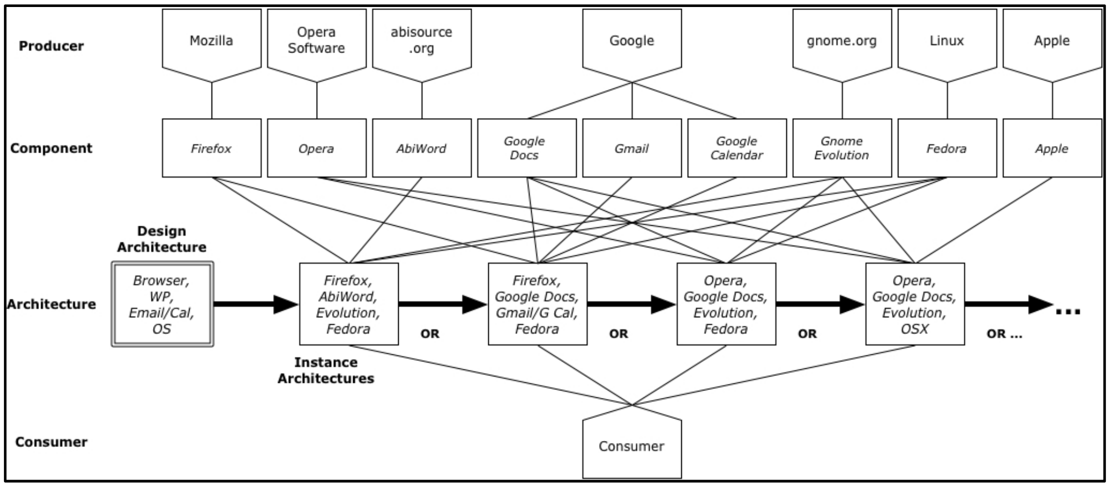


**Figure 2. A Secure Software Product Line for a Sample Enterprise Information System Consisting of Alternative, Functionally Similar Web Browser, Word Processor, Email, and Calendaring Components Sourced From Different Producers**
(Scacchi & Alspaugh, 2012)

Third, if the SPL is based on an OA that integrates software components from multiple vendors or OSS components that are subject to heterogeneous licenses and property rights, then we have the situation analogous to what we have presented in our previous work (Alspaugh, Asuncion, & Scacchi, 2009a, 2009b, 2010; Scacchi & Alspaugh, 2011, 2012a). This leads us to conclude that SPL concepts are compatible with secure OA systems that are composed from heterogeneously licensed components. Consequently, in this proposed effort, we seek to systematically investigate, model, and analyze how this might work in both (a) enterprise information systems and (b) command-and-control or related weapons systems, if they incorporate OSS and non-OSS components subject to different IP and security requirements. Our goal is to demonstrate what is possible and feasible in articulating and tailoring such an acquisition process framework, as well as demonstrating how feasible OA system alternatives within a reusable SPL facilitate cost reduction opportunities and workforce amplification capabilities in system acquisition efforts.

### *Research Problem: Articulating and Streamlining a Process for Identifying and Reviewing the Security of OA Software Systems*

As already noted in our previous work (Alspaugh, Asuncion, & Scacchi, 2009a, 2009b; Alspaugh, Scacchi, & Asuncion, 2010), software component licenses represent a collection of rights and obligations for what can or cannot be done with a licensed software component. Licenses thus denote non-functional requirements that apply to a software system or system components as intellectual property (IP) during their development and deployment. But rights and obligations are not limited to concerns or constraints applicable only to software as IP. Instead, they can be written in ways that stipulate non-functional requirements of different kinds. Consider, for example, that desired or necessary software system security properties can also be expressed as rights and obligations addressing system confidentiality, integrity, accountability, system availability, and assurance (Breaux & Anton, 2008).

Traditionally, developing robust specifications for non-functional software system security properties in natural language often produces specifications that are ambiguous, misleading, and inconsistent across system components and lacking sufficient details (Polydys & Wisseman, 2008; Yau & Chen, 2006). Using a semantic model to formally specify the security rights and obligations required for a software system or its components (Breaux & Anton, 2008; Yau & Chen, 2006) means that it may be possible to develop both a "security architecture" notation and model specification that associates given security rights and obligations across a software system or system of systems. Similarly, it suggests the possibility of developing computational tools or interactive architecture development environments that can be used to specify, model, and analyze a software system's security architecture at different times in its acquisition and development—design-time, build-time, run-time, and post-deployment support-time.

The approach we have been developing for the past few years for modeling and analyzing software system license architectures for OA systems (Alspaugh, Asuncion, & Scacchi, 2009a, 2009b, 2010; Scacchi & Alspaugh, 2012a) may therefore be extendable to also being able to address OA systems with heterogeneous software security policies, or as we have called them, *software security policy and license* rights and obligations. Furthermore, the idea of common or reusable software security policy and licenses may be analogous to the reusable security requirements templates proposed by Firesmith (2004) at the SEI. Consequently, continuing our exploration and extension of the semantic software architectural modeling, meta-modeling, and computational analysis tools to also support secure OA systems introduced in Scacchi and Alspaugh (2011, 2012) is the next stage of our research studies. With this in mind, we turn to provide additional background that helps shapes our proposed research effort.

## R&D Basis for Our Study and Approach

Across the three military services within the DoD, OA means different things and is seen as the basis for realizing different kinds of outcomes (Scacchi & Alspaugh, 2008). Thus, it is unclear whether the acquisition of a software system that is required to incorporate an OA, as well as utilize OSS technology and development processes for one military service will realize the same kinds of benefits anticipated for OA-based systems by another service. Somehow, DoD acquisition program managers must make sense of or reconcile such differences in expectations and outcomes from OA strategies in each service and across the DoD. But there is now more explicit guidance for how best to develop, deploy, and sustain complex software-intensive military systems utilizing OA and OSS components (Department of Defense Open Systems Architecture [DoDOSA], 2011; Hissam, Weinstock, & Bass, 2010).

Security is an essential issue in military software acquisition. However, we have found little effective guidance for addressing it in ways that can take advantage of the characteristics of OA SPL systems or that address the specific challenges that arise for them, such as rapid evolution, components from a variety of sources with new versions at different times, and the specific structural characteristics of OA and SPL systems.Thus, there is an essential need for knowledge and process guidance that program managers and others in the acquisition workforce can readily use in a transparent manner. Further, we anticipate that with growing awareness of emerging cyber warfare threats, the security of OA systems will potentially be mandated and thus become part of program acquisition processes. This in turn raises concern about potential cost growth and whether the acquisition workforce is well prepared to provide the needed oversight, review, and approval.

The Software Assurance Acquisition Working Group's extensive report (Polydys & Wisseman, 2007) makes clear how important security is in software acquisition; it is mentioned on most pages of the report. The recommended approaches for improving the security of software systems involve manual reviews and process improvements. Reviews by experts are recommended and discussed for requirements, architectures, components, tests, and so forth, with a new review required for each new version; we believe this could constitute a serious and time-consuming burden, especially in the context of tight budgets, short timelines, and reductions in the acquisition workforce. However, no specific guidance is offered for OA or SPL systems by these reports.

Carnegie Mellon University's Software Engineering Institute (SEI) is prominent in SPL research and practical guidance. The current Framework for Software Product Line Practice (Northrop & Clements, 2007) mentions security as one of the desired quality goals or attributes, along with reliability, usability, and others, but offers no specific guidance for addressing it. Recent papers from SEI presented at the Acquisition Research Symposium (Bergey & Jones, 2010; Jones & Bergey, 2011) discuss the benefits of a SPL strategy but do not explain how security fits into the specification or documentation of OAs.

There appears to have been comparatively little research published on the topic of security and software product lines, even without considering open architecture, and very little on security and open architecture. One of the stronger examples (Mellado, Fernández-Medina, & Piattini, 2010) summarizes a line of research over four years that addresses security requirements for software product lines (but not OAs), and guides requirements activities with the goal of addressing security concerns in a way that corresponds to the characteristics of a software product line. Their approach is process-oriented, supported by a research tool that manages the various repositories of information that are developed, but informal and primarily manual and dependent on expert practitioners. Only the requirements phase of development is addressed.

The current guidance for program managers acquiring OA systems (DoDOSA, 2011) points to the need to identify and review the use of "security engines" that can support security enforcement tasks within system development or deployment. Similarly, current guidance on best practices on improving cost effectiveness in program acquisition ("What is Better Buying Power?", 2013) offers no clear directions for how best to address or manage specific cost issues that arise during secure OA system acquisition. However, recent acquisition research indicates there is also a need for a more articulate and streamlined process that acquisition workers can follow to ensure that all relevant aspects of OA system security have been addressed in an easy to review format (cf. Scacchi & Alspaugh, 2012). Similarly, current research further points to the need to address how software reuse (Mactal & Spruill, 2012) and testing processes (Berzins, 2012) when SPLs are employed in OA systems as cost reduction and quality improvement strategies.

Most of the guidance to date for acquisition of secure OA SPLs may be summarized as follows: collect experts in security requirements, architecture, and tests, have them review and guide the manual/informal development of requirements, architecture, and tests for the product line, and repeat all or selected parts of the process for each new version and product line instance. We find little guidance for incorporating formality, effective use of software tools, and then addressing and taking advantage of the specific characteristics of OA and SPL. Similarly, there is little guidance for what the best process is to identify and review OA system security when incorporating OSS components and other reusable SPL elements. There is no guidance for how to adapt or streamline such a process to reduce costs of acquiring different kinds of systems (e.g., enterprise information systems, command and control systems, embedded weapon systems), nor what information to consider or knowledge to acquire to enable a more effective acquisition workforce.

Consequently, this leads us to consider the following questions. What is the most effective way to articulate a process for the most cost-effective acquisition of secure OA systems that can be streamlined to the needs of specific kinds of reusable software systems? What issues or research questions for acquisition research follow from such a problem? What research approach can best explore the opportunities for acquisition research built from related research efforts in OA, reusable SPL, and software architectural analysis that can also inform future acquisition cost reduction practices and improve acquisition workforce capabilities? We now turn to briefly elaborate these questions in turn through the remainder of this proposal.

### Issues for Acquisition Research

Based on current research into the acquisition of secure OA systems with OSS components and reusable SPLs (Scacchi & Alspaugh, 2011, 2012), this research project also explores and answers the following kinds of research questions: How does the interaction of requirements and architectures for secure OA systems incorporating OSS components facilitate or inhibit acquisition processes over time? What are the best available processes for continuously verifying and validating the functionality, correctness, openness, and security of OA when OSS components and SPLs are employed? How can use of continuously evolving OSS within a reusable OA or SPL be combined with the need to verify and validate critical systems security requirements and to manage their evolution? How do reliability and predictability trade off against the cost and flexibility of a secure OA system when incorporating reusable SPL components? How should secure OA software systems be developed and deployed to support warfighter modification in the field or participation in post-deployment system support, when OSS components are employed?

### Inter-Project Research Coordination

We believe we are extremely well positioned to leverage our current research work and results (Scacchi & Alspaugh, 2008, 2011, 2012a, 2012b; Alspaugh, Asuncion, & Scacchi, 2009a, 2009b, 2009c; Scacchi, Alspaugh, & Asuncion, 2010) with the effort proposed here. We build on our current research efforts in OSS (Scacchi, 2007, 2010) and software requirements–architecture interactions (Scacchi & Alspaugh, 2008; Scacchi, 2009), as well as our track record in prior acquisition research studies. Similarly, we find current related research supported by the DoD addressing related issues in OSS (Hissam, Weinstock, & Bass, 2010) also influences our proposed effort. In addition, our effort builds from and contributes to research on software system acquisition within the DoD, focusing on software reuse (Mactal & Spruill, 2012), SPLs (Guertin & Clements, 2010; Bergey & Jones, 2010), open innovation and emerging software component markets (Guertin & Womble, 2012), efficient testing of component-based OA systems and SPLs (Berzins, 2012), and how to improve software system acquisition through workforce upgrades and government–industry teaming (Heil, 2010). We thus believe our complementary research places us at an extraordinary advantage to conduct the proposed study that addresses a major strategic acquisition goal of the DoD and the military services (DoDOSA, 2011; Robert, 2012).

### Prospects for Longer-Term Acquisition-Related Research and Application

The military services have committed to orienting their major system acquisition programs around the adoption of an OA strategy that in turn embraces and encourages the adoption, development, use, and evolution of OSS (DoDOSA, 2011). Thus, it would seem there is a significant need for sustained research that investigates the interplay and inter-relationships between (a) current and emerging guidelines for the acquisition of software-intensive systems within the DoD community (including contract management and software development issues), and (b) how secure, reusable software product lines that employ an OA incorporating OSS products and production processes are essential to improving the effectiveness of future, software-intensive program acquisition efforts. Beyond this, we have begun to be invited to participate within the Defense Information System Agency (DISA) and intelligence community (IC) working groups who are formulating future OA software licensing guidelines for future system acquisition efforts. Thus, our research supported by the Acquisition Research Program is finding audiences within the DoD and other government agencies, and is on paths that can lead to improvements and cost reductions in the acquisition of software-intensive OA systems.

## Research Results

Our research studies and results are included in the remaining chapters of this final report as six individual research publications. These include four refereed conference papers and one additional paper submitted for review.

The publication venue and citation for each of these five papers appear in order of publication date as follows:

- Scacchi, W., & Alspaugh, T. A. (2013a). Streamlining the process of acquiring secure open architecture software systems, in *Proceedings of the 10th Acquisition Research Symposium,* pp. 608–623, May 2013. This paper presents the overall results from our study in 2012–2013 as applied to the development of cost-effective OA software systems composed from components sourced from online storefronts, such as those being investigated by the C4ISR research program at the Space and Naval Warfare Systems Center Pacific, San Diego, CA, or Forge.mil.
- Scacchi, W., & Alspaugh, T .A. (2013b). Processes in securing open architecture software systems, in *Proceedings of the 2013 International*

*Conference on Software and System Processes*, pp. 126–135, May 2013, San Francisco, CA. This paper closely examines technical issues that arise during the development processes for OA software systems when composing systems from components using widely available OSS development tools or methods.

- Scacchi, W., & Alspaugh, T. A. (2013c). Challenges in the development and evolution of secure open architecture command and control systems, in *Proceedings of the 18th International Command and Control Research and Technology Symposium*, Paper 098, Alexandria, VA, June 2013. This paper examines the efficacy of the development of component-based OA software systems when sourced from a multi-party engineering (MPE) effort, such as is now being considered for future C2 systems by the DoD and other government agencies.
- Alspaugh, T. A, & Scacchi, W. (2013a). Ongoing software development without classical requirements, in *Proceedings of the 21st IEEE International Conference on Requirements Engineering*, Rio de Janeiro, Brazil, pp. 165–174, July 15–19, 2013. This paper investigates how a move towards the development of component-based OA software systems may change requirements specification processes from being elicitation-oriented towards being more directed by available functional capabilities provided by commercially available software components.
- Alspaugh, T. A, & Scacchi, W. (2013b). Moving towards formalizable security licenses, at *35th International Conference on Software Engineering*, New Ideas and Emerging Results Track, San Francisco, CA (submitted for publication). This paper further elaborates properties and principles that can give rise to more streamlined ways and means for specifying the security of OA software systems, particularly those composed from commercially available software components.

Overall, we are grateful for the support and funding we have received that enabled our acquisition research to continue, and to be documented in this final report.

## Acknowledgements

Preparation of this report and all work products therein benefitted from a research grant, #N00244-12-1-0067 from the Acquisition Research Program at the Naval Postgraduate School, Monterey, CA. The lead author, Scacchi, also acknowledges research support from the National Science Foundation under grant #1256593. None of the content of this final report has been reviewed, approved, or endorsed by the ARP, NPS, U.S. Navy, Department of Defense or any other government agency. The work presented is solely the responsibility of the authors.

## References

Alspaugh, T. A, Asuncion, H., & Scacchi, W. (2009a). Software licenses, open source components, and open architectures. In *Proceedings of the Sixth Annual Acquisition Research Symposium* (NPS-AM-09-026), Naval Postgraduate School, Monterey, CA, May.

Alspaugh, T. A, Asuncion, H., & Scacchi, W. (2009b). Intellectual property rights requirements for heterogeneously licensed systems. In *Proceedings of the 17th International Conference on Requirements Engineering (RE09*), Atlanta, GA, pp. 24–33.

Alspaugh, T. A, & Scacchi, W. (2013a). Ongoing software development without classical requirements. In *Proceedings of the 21st IEEE International Conference on Requirements Engineering*, Rio de Janeiro, Brazil, pp. 165–174.

Alspaugh, T. A, & Scacchi, W. (2013b). *Moving towards formalizable security licenses* (submitted for publication).

Alspaugh, T. A., Scacchi, W., & Asuncion, H. (2010). Software licenses in context: The challenge of heterogeneously licensed systems, *Journal of the Association for Information Systems, 11*(11), 730–755.

What is Better Buying Power? (2013). Retrieved from http://bbp.dau.mil/

Bergey, J., & Jones, L. (2010). Exploring acquisition strategies for adopting a software product line approach. In *Proceedings of the Seventh Annual Acquisition Research Symposium* (Vol. 1, pp. 111–122), Naval Postgraduate School, Monterey, CA.

Berzins, V. (2012). Certifying tools for test reduction in open architecture. In *Proceedings of the Ninth Annual Acquisition Research Symposium* (Vol. 1, pp. 185–194), Naval Postgraduate School, Monterey, CA.

Bosch, J. (2000). *Design and use of software architectures: Adopting and evolving a product-line approach*. New York, NY: Addison-Wesley Professional.

Breaux, T. D., & Anton, A.I. (2008). Analyzing regulatory rules for privacy and security requirements. *IEEE Transactions on Software Engineering, 34*(1), 5–20.

Clements, P., & Northrop, L. (2003). *Software product lines: Practices and patterns*. New York, NY: Addison-Wesley Professional.

Department of Defense Open Systems Architecture (DoDOSA). (2011, December). *Contract guidebook for program managers* (Vol. 0.1). Retrieved from https://acc.dau.mil/OSAGuidebook

Firesmith, D. (2004, Jan–Feb). Specifying reusable security requirements. *Journal of Object Technology, 3*(1), 61–75.

Guertin, N., & Clements, P. (2010). Comparing acquisition strategies: Open architecture versus product lines. In *Proceedings of the Seventh Annual Acquisition Research Symposium* (Vol. 1, pp. 78–90), Naval Postgraduate School, Monterey, CA.

Guertin, N., & Womble, B. (2012). Competition and the DoD marketplace. In *Proceedings of the Ninth Annual Acquisition Research Symposium* (Vol. 1, pp. 76–82), Naval Postgraduate School, Monterey, CA.

Heil, J. (2010). Enabling software acquisition improvement: Government and industry software development team acquisition model. In *Proceedings of the Seventh Annual Acquisition Research Symposium* (Vol. 1, pp. 203–218), Naval Postgraduate School, Monterey, CA.

Hissam, S., Weinstock, C. B., & Bass, L. (2010). On open and collaborative software development in the DoD. In *Proceedings of the Seventh Annual Acquisition Research Symposium* (Vol. 1, pp. 219–235), Naval Postgraduate School, Monterey, CA.

Jones, L., & Bergey, J. (2011). An architecture-centric approach for acquiring software-reliant system. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (Vol. 1), Naval Postgraduate School, Monterey, CA.

Mactal, R., & Spruill, N. (2012). A framework for reuse in the DoN. In *Proceedings of the Ninth Annual Acquisition Research Symposium* (Vol. 1, pp. 149–164), Naval Postgraduate School, Monterey, CA.

Mellado, D., Fernández-Medina, E., & Piattini, M. (2010). Automated support for security requirements engineering in software product line domain engineering. *Information and Software Technology, 52*(10), 1094–1117.

Naegle, B., & Petross, D. (2008). Software architecture: Managing design for achieving warfighter capability. In *Proceedings of the Fifth Annual Acquisition Research Symposium* (NPS-AM-07-104), Naval Postgraduate School, Monterey, CA.

Polydys, M. L., & Wisseman, S. (2008). Software assurance in acquisition: Mitigating risks to the enterprise. Retrieved from https://buildsecurityin.us-cert.gov/swa/acqact.html

Robert, J. (2012). Software strategy for the defense enterprise. In *Proceedings of the Ninth Annual Acquisition Research Symposium*, Naval Postgraduate School, Monterey, CA.

Scacchi, W. (2007). Free/open source software development: Recent research results and methods. In M. Zelkowitz (Ed.), *Advances in Computers*, *69*, 243–295.

Scacchi, W. (2009). Understanding requirements for open source software. In K. Lyytinen, P. Loucopoulos, J. Mylopoulos, and W. Robinson (Eds.), *Design requirements engineering: A ten-year perspective* (pp. 467–494), LNBIP 14, Springer Verlag.

Scacchi, W. (2010). The future of research in free/open source software development. In *Proceedings of the ACM Workshop on the Future of Software Engineering Research (FoSER)* (pp. 315–319), Santa Fe, NM.

Scacchi, W., & Alspaugh, T. A. (2008). Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In *Proceedings of the Fifth Annual Acquisition Research Symposium* (NPS-AM-08-036), Naval Postgraduate School, Monterey, CA.

Scacchi, W., & Alspaugh, T. A. (2011). Advances in the acquisition of secure systems based on open architectures. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (Vol. 1), Naval Postgraduate School, Monterey, CA.

Scacchi, W., & Alspaugh, T. A. (2012a). Understanding the role of licenses and evolution in open architecture software ecosystems, *Journal of Systems and Software, 85*(7), 1479–1494.

Scacchi, W., & Alspaugh, T. A. (2012b). Addressing challenges in the acquisition of secure software systems with open architectures. In *Proceedings of the Ninth Annual Acquisition Research Symposium* (Vol. 1, pp. 165–184), Naval Postgraduate School, Monterey, CA.

Scacchi, W., & Alspaugh, T. A. (2013a). Streamlining the process of acquiring secure open architecture software systems. In the *Proceedings of the 10th Annual Acquisition Research Symposium* (pp. 608–623), Naval Postgraduate School, Monterey, CA.

Scacchi, W., & Alspaugh, T. A. (2013b). Processes in securing open architecture software systems In *Proceedings of the 2013 International Conference on Software and System Processes* (pp. 126–135), San Francisco, CA.

Scacchi, W., & Alspaugh, T. A. (2013c). Challenges in the development and evolution of secure open architecture command and control systems. In *Proceedings of the 18th International Command and Control Research and Technology Symposium*, Paper 098, Alexandria, VA.

Scacchi, W., & Alspaugh, T. A. (2013d). Advances in the acquisition of secure systems based on open architectures, *Journal of Cybersecurity & Information Systems (*forthcoming).

Northrop, L., & Clements, P. (2007). *A framework for software product line practice,* Version 5.0. http://www.sei.cmu.edu/productlines/frame_report/index.html

Womble, B., Schmidt, W., Arendt, M., & Fain, T. (2011). Delivering savings with open architecture and product lines. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (Vol. 1), Naval Postgraduate School, Monterey, CA.

Yau, S. S., & Chen, Z. (2006). A framework for specifying and managing security requirements in collaborative systems. In *Proceedings of the Third International Conference on Autonomic and Trusted Computing (ATC 2006)* (pp. 500–510).

# Streamlining the Process of Acquiring Secure Open Architecture Software Systems

## Abstract

We present results from our ongoing investigation of how best to acquire secure open architecture (OA) software systems. These systems incorporate software product line (SPL) practices that include closed source proprietary software and open source software (OSS) components, where such components and overall system configurations are subject to different security requirements. The combination of SPLs and OSS components within secure OA systems represents a significant opportunity for reducing the acquisition costs of software-intensive systems. We seek to make this a simpler, more transparent, and more tractable process. Such a process must be easy to reuse, adapt, and streamline for different system application domains in order to realize cost reductions and improve acquisition workforce capabilities. Further, such a process should be aligned with Better Buying Power initiatives addressing OA systems, improved competition, defense affordability, and acquisition workforce improvements. We identify different ways and means for streamlining the acquisition process for secure OA software systems through a focus on doing more with limited resources. Along the way, we pay particular attention to revealing how software licensing practices can affect cost in ways that hamper or better the buying power of acquisition programs.

## Introduction

Our focus in this effort is to identify ways and means for streamlining the acquisition process for secure OA systems. These OA systems often rely on the integration of components that are independently developed by different software producers and made available as either OSS or proprietary closed source software executables. Program managers, acquisition officers, and contract managers will increasingly be called on to review and approve security measures employed during the design, implementation, and deployment of OA systems (Department of Defense Open Systems Architecture [DoDOSA], 2011). Our effort builds on both our prior acquisition research (e.g., Scacchi & Alspaugh, 2008, 2011, 2012a), and related acquisition research efforts at the PEO IWS (Guertin & Clements, 2010; Guertin & Womble, 2012, Womble, Schmidt, Arendt, & Fain, 2011), Department of the Navy (Mactall & Spruill, 2012), and Software Engineering Institute (SEI) that address SPLs (Bergey & Jones, 2010; Jones & Bergey, 2011; Northrop & Clements, 2007). It is also influenced by related research in the DoD community addressing OSS (Defense Information Systems Agency [DISA], 2012; Hissam, Weinstock, & Bass, 2010; Kenyon, 2012; Martin & Lippold, 2011), component-based software ecosystems (Reed, Benito, Collens, & Stein, 2012; Scacchi & Alspaugh, 2012b), and Better Buying Power initiatives (Defense Acquisition University [DAU], 2012).

OSS represents an integrated web of people, processes, and organizations, including project teams operating as virtual organizations (Scacchi, 2007, 2009, 2010). There is a basic need to understand how to identify an optimal mix of OSS within OA systems as products, production processes, practices, community activities, and multi-project (or multi-organization) software ecosystems. However, the relationship among OA, OSS, security requirements, and acquisition is poorly understood (cf. Naegle & Petross, 2007; Scacchi, 2009, 2010, 2011; Scacchi & Alspaugh, 2011, 2012b). Subsequently, in 2007–08, we began by examining how different OSS licenses can encumber software systems with OA, which therefore give rise to new requirements for how best to acquire software-intensive systems with OA and OSS elements (Scacchi & Alspaugh, 2008).

As a result of our recent acquisition research efforts, we have been able to demonstrate that it is both possible and feasible to develop OA systems that incorporate best-of-breed software components, whether proprietary or OSS, in ways that can reduce the initial and sustaining acquisition costs of such systems.

We believe that such results are applicable to both enterprise information systems, which are widespread throughout the DoD and the U.S. government, as well as command and control (C2; e.g., Reed et al., 2012; Scacchi & Alspaugh, 2013b; Scacchi, Brown, & Nies, 2012) and other defense systems. Doing so however requires new guidance, and ideally, automated tools, for explicitly modeling and analyzing the architecture of an OA system during its development and evolution, along with modeling the annotating the architecture with software component license rights and obligations. Our results thus demonstrate a major technological advance in the acquisition and development of OA systems, as a breakthrough in simplifying software license analyses throughout the contracting activities. Creating similar advances for streamlining the acquisition process, while reducing the costs of secure OA systems, is the next breakthrough that is needed.

In this paper, we describe ways and means for articulating, tailoring, and streamlining the process for how to simply and transparently specify and assess OA system security when acquiring different kinds of OA systems, and to do so in ways that highlight opportunities for cost reduction through system security requirements specification and OA system acquisition process streamlining. We provide examples of complex software elements that are applicable to many kinds of software-intensive systems within the DoD, as well as within other government agencies and industrial firms. But we start in the next section by reiterating Better Buying Power principles and initiatives that guide this research by focusing on how to promote competition in acquisition and development of secure OA systems.

## OA and Better Buying Power

Better Buying Power (http://bbp.dau.mil/) is part of the DoD's mandate to do more without more by implementing best practices in acquisition. BBP identifies seven areas of focus that group a larger set of 36 initiatives that offer the potential to restore affordability in defense procurement and improve defense industry productivity. One of the seven areas focuses on promoting competition, and this area includes an initiative to "enforce open system architectures and effectively manage technical data rights" (DAU, 2012). Technical data rights pertain to two categories of intellectual property (IP): They refer to the government's rights to (a) technical data (TD–e.g., product design data, computer databases, computer software documentation) and (b) computer software (CS–e.g., source code, executable code, design details, processes, and related materials). These rights are realized through IP licenses provided by system product or service providers (e.g., software producers) to the government customer, so long as the customer fulfills the obligations stipulated in the license agreement (e.g., to indicate how many software users are authorized to use the licensed product or service according to a fee paid). As already noted, our acquisition research has focused on issues addressing OA systems and IP licenses since 2008 (Scacchi & Alspaugh, 2008).

OA software systems offer the potential to improve acquisition by providing new ways and means to acquire, develop, deploy, and sustain software-intensive systems. These new ways and means in turn may transform how the DoD acquires complex systems by moving away from long-duration, proprietary (closed) system architecture, and difficult-to-control-cost of system development efforts, towards systems that may be more rapidly assembled and integrated in an OA manner with more transparent costs. Such a transformation may in turn reduce vendor lock-ins that often are associated with rising costs to sustained deployed systems that are inaccessible to competing vendors. So, closed architecture legacy systems often are subject to IP licenses whose consequence is to reduce competition while increasing system sustainability costs. Our research on OA systems dating many years back (Scacchi & Alspaugh, 2008) has consistently been aligned with efforts for improving competition in software system development and evolution through investigation of innovative ways and means to acquire and develop component-based OA software systems that are subject to diverse, heterogeneous IP licenses (Alspaugh, Scacchi, & Asuncion, 2010). But there is more to do to improve competition and defense affordability while effectively managing TD rights when addressing the acquisition of secure OA systems. In particular, this includes understanding that the processes for acquiring such systems are facilitated or constrained in light of overall BBP guidance and best practices, as well as how best to improve and streamline these processes. These topics are our focus in the remainder of this chapter.

## How BBP Impacts the Processes for Acquiring OA Systems

The move to OA systems represents a transition from the acquisition of monolithic systems to the acquisition of reusable system components that can be integrated to realize different configurations of a software product family for a specific application domain (Bergey & Jones, 2010; Guertin & Clements, 2010; Jones & Bergey, 2011; Northrop & Clements, 2007; Reed et al., 2012; Scacchi & Alspaugh, 2012b; Womble et al., 2011). These components are acquired within a software ecosystem that is evolving towards component provisioning within open repositories, where components from different producers are available for selection, evaluation, and system integration (Guertin & Womble, 2012; Martin & Lippold, 2011; Reed et al., 2012; Scacchi, 2007; Scacchi & Alspaugh, 2012, 2013b). Figure 1 provides a graphic view of how such an ecosystem spans from a sample of software producers and components through system integrators to software consumers and users.

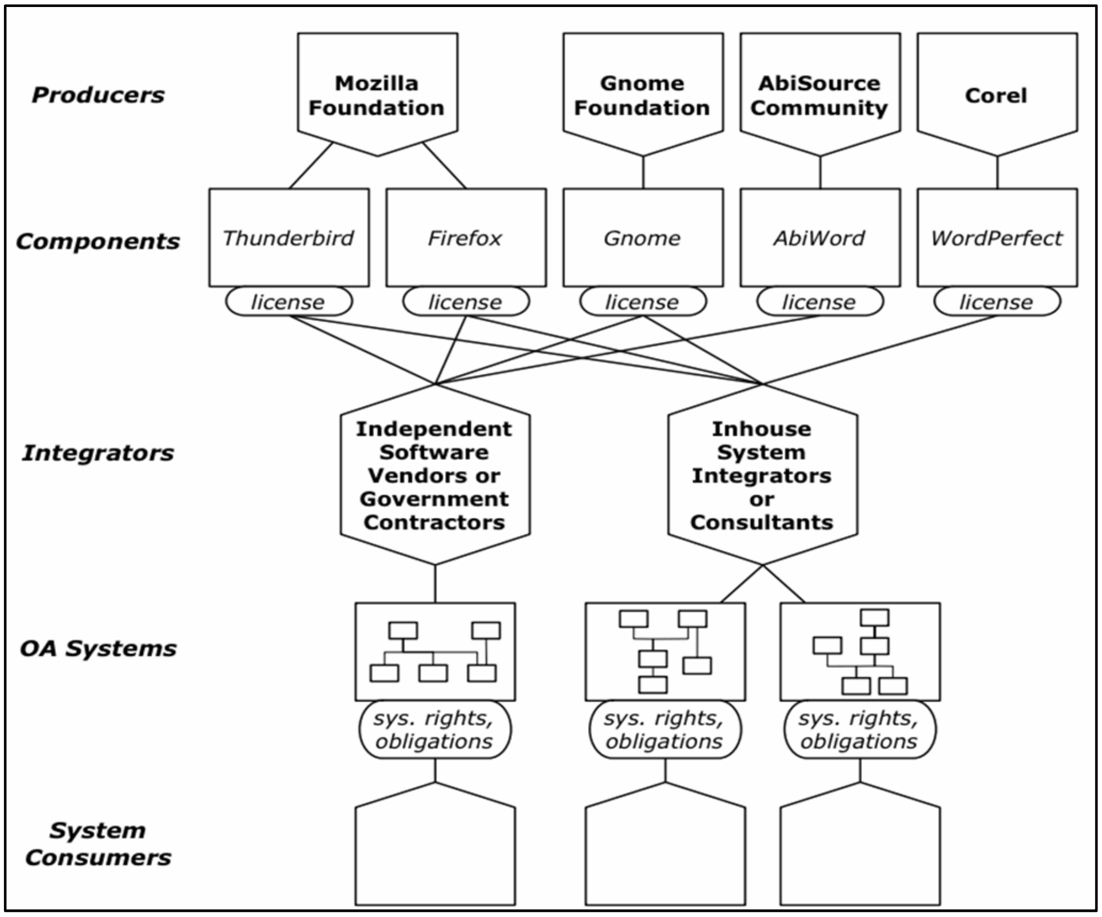


**Figure 1. A Sample Software Ecosystem of Producers, Components, Integrators, Alternative OA Systems, and Consumers/Users**
(Scacchi & Alspaugh, 2012b)

Figure 2 provides a view of a sample of lightweight software components ("widgets" targeted for software developers or integrators in this example) for download and installation within a Web browser. These widgets, made by different producers, are available for acquisition from Google's Chrome Web Store.

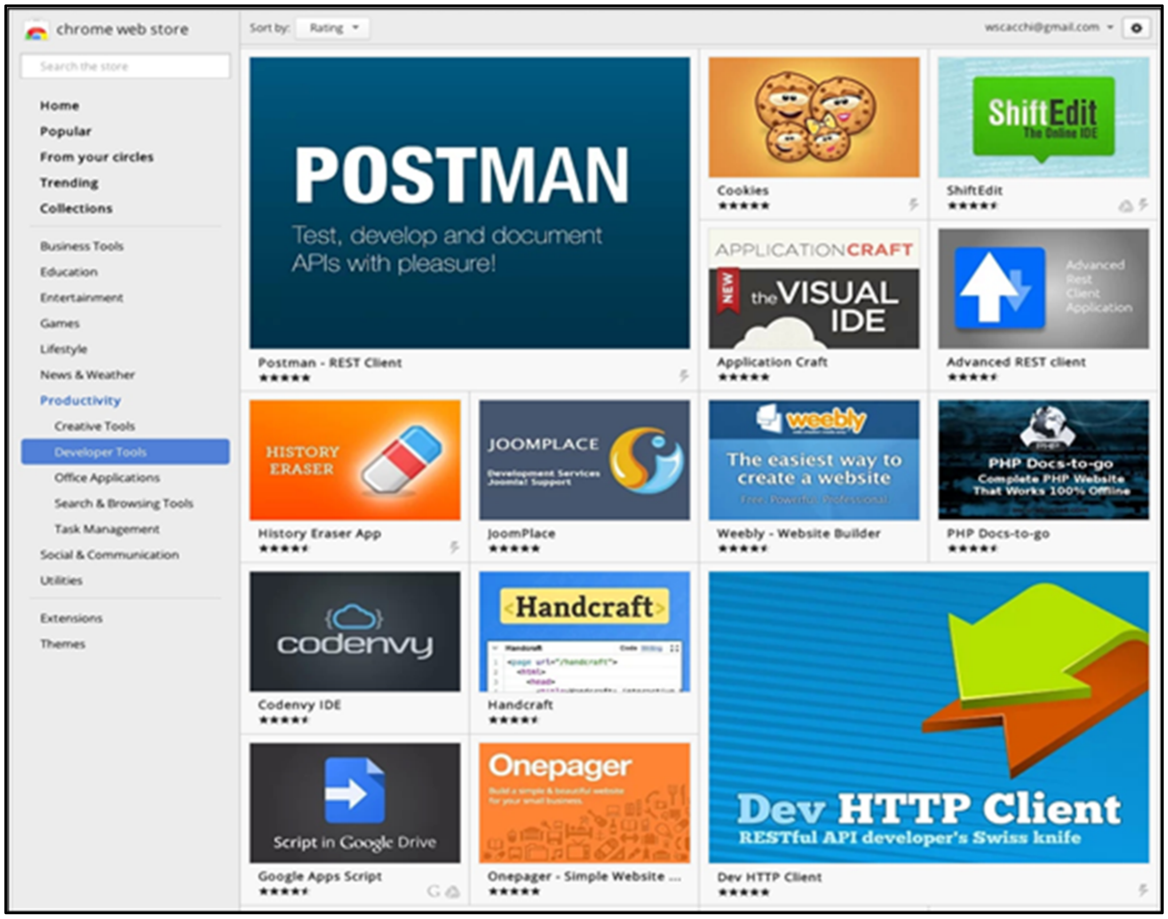
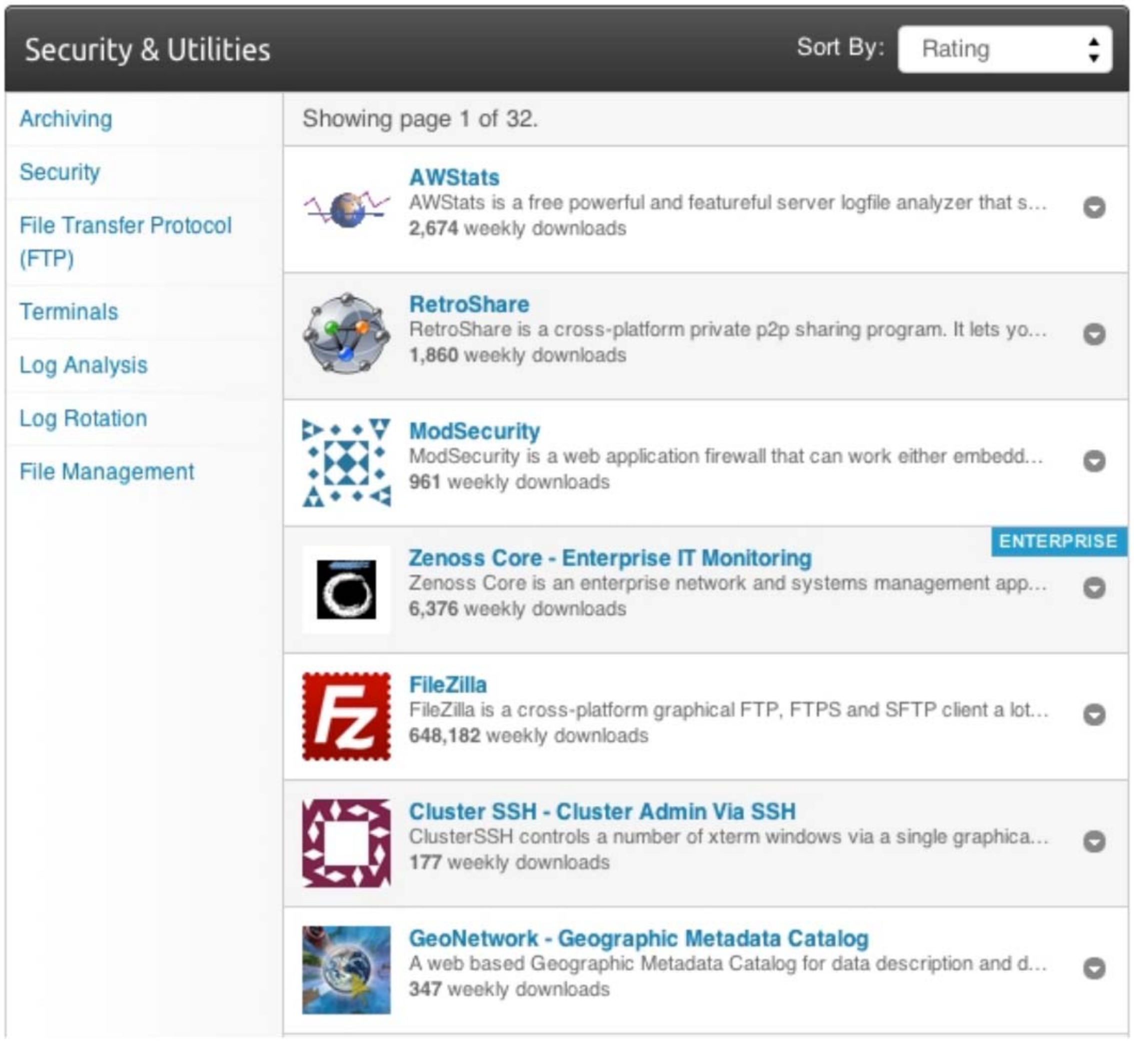


**Figure 2. A Sample View of Lightweight Software Components ("Widgets") That Can be Readily Acquired for Evaluation or Integration from Google's Chrome Web Store**

Such an online store serves as a marketplace that provides access to ready-to-run, closed source software executables from within an online software repository that can be navigated using the menu on the left-side, browsed by scrolling, or by entry of a search term or phrase in the upper-left corner (see Figure 2).

Software components in an online marketplace like this are rated or recommended by other consumers, but the IP licenses for the TD and CS are hidden away with each component and may be challenging to locate prior to installation. Google Play for Android apps and the Apple App Store also offer software (widget) components for their respective computing platforms (Android and iPhone smartphones, or Nexus and iPad mobile tablet computers).

Figure 3 provides a view of a different online repository that exclusively features OSS components found at SourceForge.net (similar to Forge.mil; DISA, 2012; Martin & Lippold, 2011), where the IP licenses for each software component are prominently displayed when one selects to look more closely into the details and development status of a component of interest. In contrast to the web browser–specific software widgets available at the Chrome Web Store, the OSS components at SourceForge.net represent more substantial, production-oriented software tools or utilities that can operate as standalone application programs. Forge.mil may be envisioned to provide support for providing access to pre-tested and certified software components, whether lightweight widgets or more substantial application systems, in OSS code and ready-to-run executable forms with TD rights designed for government purposes. Thus overall, what we see is that if we want to improve competition through the acquisition of component-based software systems, that our choice of which online repository or marketplace to use leads to different kinds of software components with different IP license schemes.



Showing page 1 of 32.

**AWStats**
AWStats is a free powerful and featureful server logfile analyzer that s...
2,674 weekly downloads

**RetroShare**
RetroShare is a cross-platform private p2p sharing program. It lets yo...
1,860 weekly downloads

**ModSecurity**
ModSecurity is a web application firewall that can work either embedd...
961 weekly downloads

**Zenoss Core - Enterprise IT Monitoring**
Zenoss Core is an enterprise network and systems management app...
6,376 weekly downloads

**FileZilla**
FileZilla is a cross-platform graphical FTP, FTPS and SFTP client a lot...
648,182 weekly downloads

**Cluster SSH - Cluster Admin Via SSH**
ClusterSSH controls a number of xterm windows via a single graphica...
177 weekly downloads

**GeoNetwork - Geographic Metadata Catalog**
A web based Geographic Metadata Catalog for data description and d...
347 weekly downloads

**Figure 3. Sample of OSS Security/Utility Components Found at SourceForge.net**

Next, we encounter challenges in the development of integrated OA systems that are configured from different software components. Figure 4 provides a visual representation that shows different software producers can develop different kinds of software components (small, medium, or large size/capability), which system integrators can select from in order to create an OA system product line of alternative component configurations.

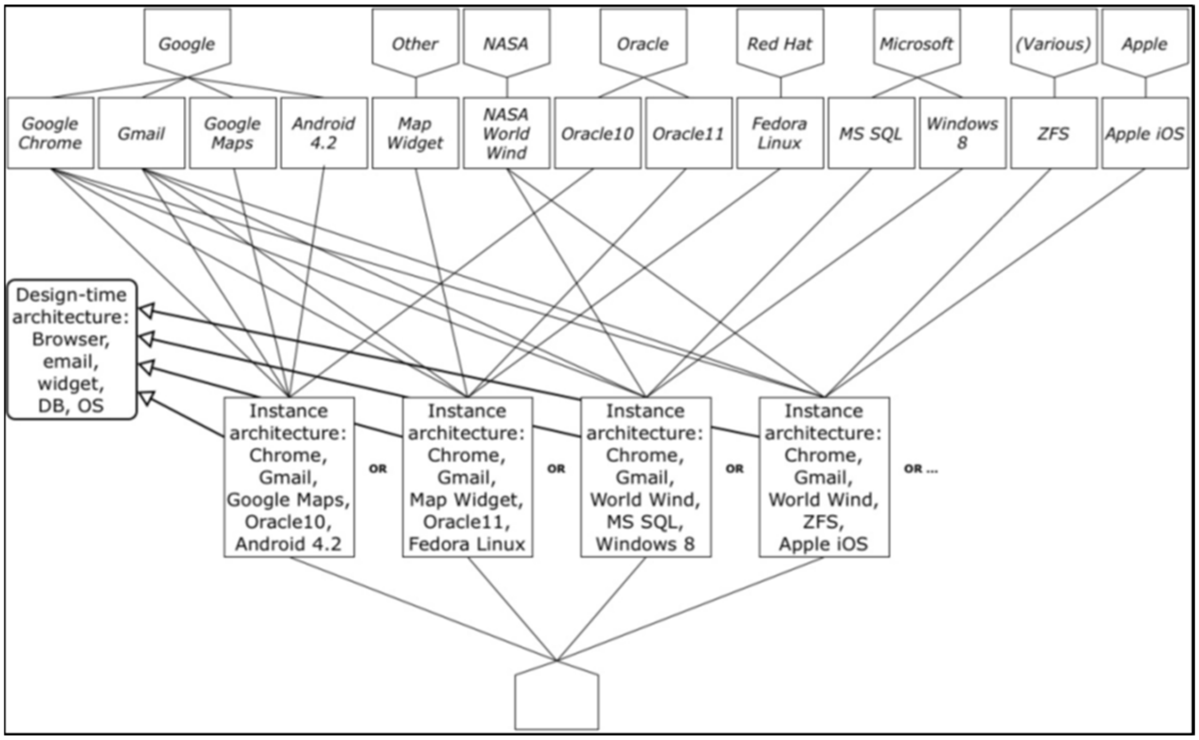

Google
Other
NASA
Oracle
Red Hat
Microsoft
(Various)
Apple
Google Chrome
Gmail
Google Maps
Android 4.2
Map Widget
NASA World Wind
Oracle10
Oracle11
Fedora Linux
MS SQL
Windows 8
ZFS
Apple iOS
Design-time architecture: Browser, email, widget, DB, OS
Instance architecture: Chrome, Gmail, Google Maps, Oracle10, Android 4.2
OR
Instance architecture: Chrome, Gmail, Map Widget, Oracle11, Fedora Linux
OR
Instance architecture: Chrome, Gmail, World Wind, MS SQL, Windows 8
OR
Instance architecture: Chrome, Gmail, World Wind, ZFS, Apple iOS
OR ...

**Figure 4. A Component-based Software Ecosystem that Configures a Product Line of Four Alternative System Configurations, Conforming to an OA System Design in Figure 5**

Figure 5 shows a simple OA system design that accommodates alternative software components as applications or infrastructure elements which may be subject to OSS or proprietary licenses. The applications ("apps") may include small, proprietary, and lightweight browser widgets, or large components like OSS-based Web browsers. The infrastructure software, which is assumed to serve as an independent foundation for application software, can include proprietary or OSS components like database management systems (or network file systems or other online repositories), and computer operating systems. Figure 6 then displays the selection of one set of conforming software components selected from the software ecosystem in Figure 4 that also conforms to the OA system design in Figure 5.

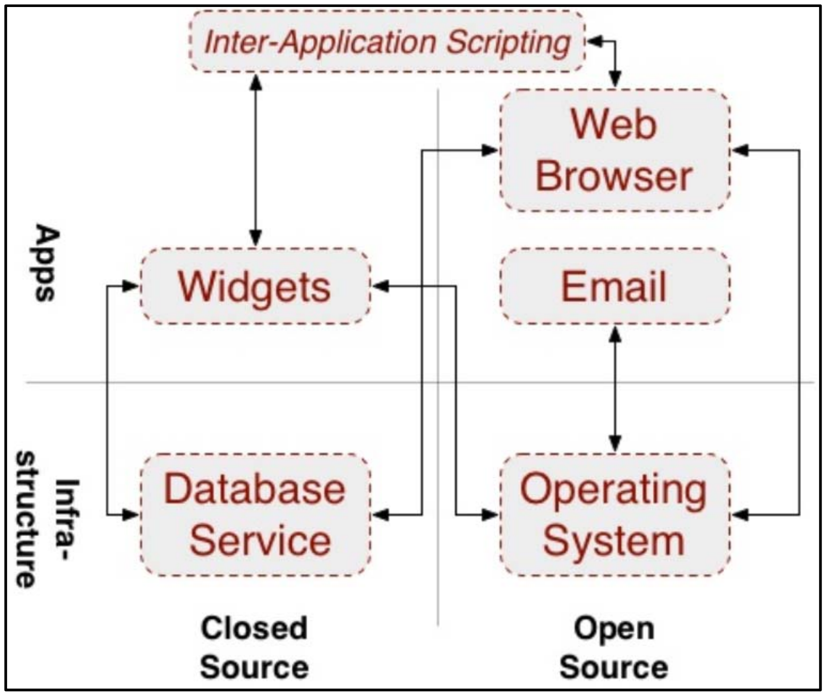


**Figure 5. A Simple OA System Design that Accommodates Software Components as Applications or Infrastructure Elements, Shown in Figure 4**

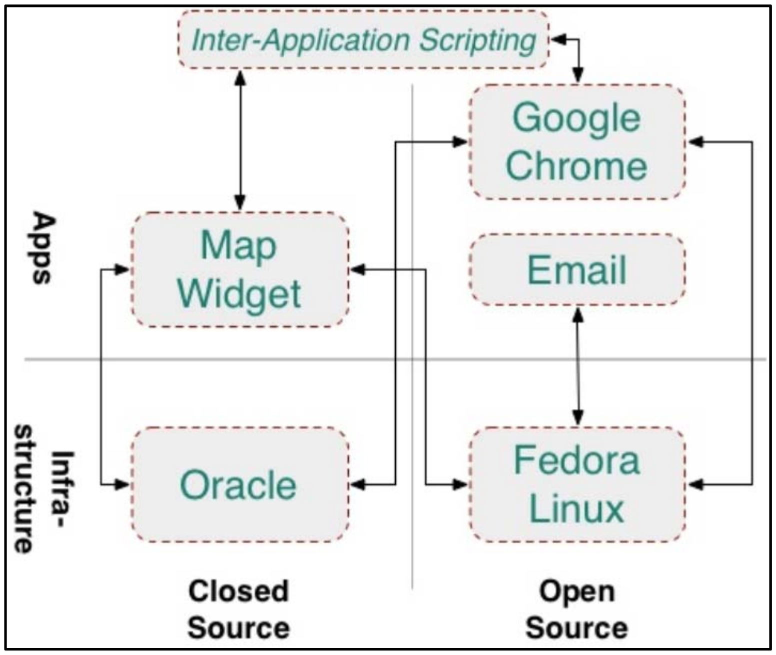


**Figure 6. A Selection of Software Components from the Ecosystem in Figure 4 Conforming to the OA System Design in Figure 5**

Lightweight software widgets are developed using domain-specific scripting languages, like JavaScript or PHP, which are designed to operate with popular Web browsers or browser-based integrated system environments. These widgets commonly represent small programs that are often produced with limited resources on short time frames and sometimes constitute only hundreds of lines of scripting source code. More complex integrated capabilities can be constructed by integrating a set of selected widgets using additional scripting code via integration techniques that produce inter-application “mashups.” Consequently, there is substantial competition in the widget/app marketplace. However, these lightweight software components often have short-term life cycles and few updates before their demise.

At present, lightweight software components tend not to be sustained for periods beyond their early availability, widespread adoption, and deployment. Their life cycle may be measured in months, rather than years (or decades). Consequently, these lightweight components are effectively designed to be disposable, low-cost software—acquire it, then use it until something better is available, then repeat. This means it may be easier for producers of such components to develop new components with new(er) capabilities, technologies, or remote services, rather than trying to sustain the short-lived legacy code. In this regard, producing new components may be less costly than maintaining legacy components that depend on technologies or services that may no longer be available or viable. Lightweight software components with short life cycles in this regard may improve competition, overall system adaptability, and affordability, while reducing vendor lock-on to costly legacy software. Updated versions of such components may be provided to repair or replace problematic implementations, but they may also appear simply as an inducement to maintain use of the component until an extended (e.g., “Pro”) version becomes available for acquisition. Finally, the globally dominant online app stores like those operated by Apple, Blackberry, Google, Microsoft, and others tend to primarily or exclusively distribute small, lightweight software components as proprietary closed source executables on a per user basis and with IP licenses that prohibit open access, reuse, modification, and redistribution. But these choices are determined by the business models of the online repository or store operators, rather than on some critical technological dependency or constraint. So, new software products like lightweight components from online repositories and stores will likely require more agile acquisition processes, contracting practices, replacement/upgrade, and IP license management regimes.

In contrast, the Web browsers in which these widgets run are themselves substantial multi-million source lines of code software components that are often integrated into larger software-intensive defense systems, like the C2RPC experimentation platform (Garcia, 2010; Gizzi, 2011). These browsers and other integrated software packages are tested and deployed on global scales, which in turn helps to insure their viability, sustainability, and quality within a highly competitive software product ecosystem. Their availability as either proprietary or

OSS forms indicates there is active, ongoing competition among their producers. In addition, these OSS browsers and other integrated software packages based on open standards (e.g., OpenOffice, LibreOffice) mean that commonly used, large-scale software applications and software infrastructure systems are available with IP licenses that offer lower acquisition costs and improved competition, as well as improved defense affordability options.

OSS components found at SourceForge.net or Forge.mil are typically somewhere in between in size, complexity, and functional capability of lightweight widgets and large integrated software packages. However, there is no requirement imposed in OSS repositories about what size, complexity, or capability components can be made available. So many OSS components range in size from thousands to hundreds of thousands of source lines of code, and they vary in terms of their quality and sustainability. OSS components from online repositories like SourceForge.net are generally available for free or at a low cost, and may or may not be designed around open standards. Many OSS-based applications do not rely on any standards, while much OSS-based infrastructure software relies on either open industry standards or de facto standards grounded in proprietary and legacy systems (sometimes referred to as “workalike” or functionally similar [Scacchi & Alspaugh, 2012b] systems). In contrast, the DoD is seeking to make sure its online OSS repositories like Forge.mil (or others) will only host components that are pre-tested and certified as compliant with relevant standards, quality/reliability indicators, and security policies relevant to their problem domain (DISA, 2012; Kenyon, 2012; Reed et al., 2012).

Software components and online component repositories/stores offer the potential to transform the ways and means for acquiring and developing component-based OA systems. But at present, the size, functional complexity, quality, extensibility, and sustainability of different software components varies in part based on the repository or store from which they are acquired. Though components that can be integrated within a secure OA system offer the potential to increase competition, the acquisition processes need to be updated and the acquisition workforce newly trained in these new ways and means, in order to maximize the likelihood for Better Buying Power initiatives addressing OA systems.

## How Best to Improve and Streamline Acquisition Processes for Secure OA Systems

The transition to the development, deployment, and sustainment of software-intensive systems based on an OA means that new or revised acquisition processes may be needed. In particular, we believe such advances call for (a) the adoption of open business models within the DoD and its industry partners, (b) open source approaches to creating Web-based acquisition processes (Scacchi, 2001) that specifically address BBP initiatives, and (c) employing techniques for streamlining these processes (Choi & Scacchi, 2001; Nissen, 1998; Scacchi, 2001; Scacchi & Noll, 1997) for secure OA systems. Each is described in turn in this section.

### Encouraging the Adoption of Open (Source) Business Models

One goal of BBP initiatives is to reduce costs by improving competition. Such a situation may be disconcerting to legacy software producers who are long experienced with the long-term development of proprietary, large-scale software systems with closed architectures that are subject to traditional, cumbersome, and costly software product licenses and license management regimes (Anderson, 2012; Konary, 2009). A move towards agile and adaptive development of secure OA systems based on software components, that can be developed and integrated more rapidly and at lower cost with more favorable IP licenses, represents a new acquisition strategy (Reed et al., 2012; Scacchi & Alspaugh, 2013b). This suggests the need to incentivize software producers and system integrators, so as to ensure their ability to effectively produce both proprietary and OSS components that are economically viable yet cost effective to the government over the life of such systems. The overall BBP mandate recognizes this situation but does not specify the means for how best to accomplish it. We believe one promising candidate is for defense enterprises and program offices to adopt new open business models.

The business models we have in mind should be rendered in an open source format. Such models should be computer-processable (i.e., amenable to automated enactment support) and transparent to participants in the acquisition workforce (e.g., available through Web-based application systems [Scacchi, 2001; Scacchi & Nissen, 1997]). They should be similarly open to participants in software producer, system integrator, and system user enterprises. These models should incorporate a product line of common/reusable open system architectures that can integrate functionally similar software components in order to realize domain-specific system solutions (e.g., for domains like command and control, weapon systems, or enterprise computing; Bergey & Jones, 2010; Guertin & Clements, 2010; Jones & Bergey, 2011; Northrop & Clements, 2007; Reed et al. 2012; Scacchi & Alspaugh, 2012b; Womble et al., 2007). These business models should incorporate Web-based computational models of acquisition processes (Nissen, 1998; Scacchi, 2001; Scacchi & Nissen, 1997) that manage the system development and support processes that surround the OA product line system models. Finally, these business models should highlight which acquisition or system development processes, or OA system features, require attention to IP licenses.

Prior research has demonstrated that significant cost reductions and process streamlining are possible when open source business process models are utilized (Choi & Scacchi, 2001; Nissen, 1998; Scacchi, 2001; Scacchi & Nissen, 1998).

These kinds of models can be subjected to performance measurement across multiple acquisition process enactments, continuous improvement, and process redesign by the acquisition workforce (Scacchi, 2001). Now we propose to enhance and extend their value through the incorporation of OA system models. While demonstrating such a capability is beyond the scope of this study, prior research results suggest the plausibility of such an approach. So future acquisition research targeting BBP may be directed to creation of open business models that can be openly accessed, reused, modified, and redistributed where appropriate.

### Open Source Models of Acquisition Processes

As noted, prior research has demonstrated the value and real payoffs of Web-based computational models for defense acquisition processes (Choi & Scacchi, 2001; Nissen, 1998; Scacchi, 2001; Scacchi & Nissen, 1998). However, many technological advances, organizational transformations, and shifting defense priorities have occurred since these results were first demonstrated and deployed years ago. Our own studies on design of secure OA system product lines are an example of technological advances not addressed in our earlier process models. But without explicit, open source process models that can be enacted through Web-based user interfaces (i.e., Web browsers accessing remote application services while tracking process enactment progress and performance parameters), then the ability to realize their benefits like process streamlining and cost reduction is elusive and difficult to manifest. Among the reasons for why this is so is overcoming gaps for how best to (a) monitor and measure acquisition process performance without automated enactment support; (b) redesign legacy processes to better accommodate technical advances and to remove ineffective bureaucratic procedures, or that transform acquisition processes in ways that do more with less while also empowering the acquisition workforce; (c) design new acquisition processes like those for acquiring secure, component-based OA software systems subject to multiple IP licenses; and (d) accommodate software IP licenses and license management regimes as acquisition process cost elements. To better understand what gaps exist in these four areas, we now describe techniques for streamlining the acquisition processes for secure OA system.

### Techniques for Streamlining Acquisition Processes for Secure OA Systems

A goal of this paper is to identify ways and means for streamlining acquisition processes for secure OA systems. In particular, we focus on four kinds of techniques that can be used to streamline such processes in ways that are responsive to the BBP initiative for open system architectures subject to complex IP licenses. These techniques are illustrative rather than exhaustive, as other kinds of techniques in other areas are also expected to exist and be available for practice by the acquisition workforce.

***Process Measurement and Assessment***—The most direct way to determine the efficiency and effectiveness of acquisition processes is by measuring their structural attributes. Such attributes indicate things such as (a) length of longest path of process steps/actions (process length); (b) number of distinct process paths (process width); (c) number of sub-process levels (process depth); (d) total number of process steps (process size); and (e) process size divided by process length (process parallelism), and other metrics (Nissen, 1998). But without an explicit graph-based model of acquisition processes, such measurements are impractical or implausible. Nonetheless, such metrics are a key for where to look for process improvement or process redesign opportunities. One might also recognize that some acquisition processes are underspecified, for example, by not explicitly accounting for where software licenses are negotiated or license trade-off analysis done. Similarly, as OA systems may include software components subject to different licenses (Alspaugh & Scacchi, 2010), then how are component–component license interactions assessed or analyzed, if at all? If acquisition processes do not explicitly account for new acquisition or license management activities that emerge due to advances in OA system development, then such processes are underspecified, which means their costs are hidden and difficult to control or minimize. Thus, if the goal of BBP is to help improve the affordability of OA systems within the DoD, then we need to be able to systematically model, measure, and assess our acquisition processes (Scacchi, 2001). Similarly, we need to better understand how to measure and assess open business models for use within the DoD and its industry partners to incentivize and continuously improve competition and defense affordability

***Process Redesign and Evolution***—Once we have the ability to measure and assess current and emerging acquisition processes for secure component-based OA systems, we can then begin to analyze (or simulate) them in ways that reveal process redesign opportunities and transformation heuristics (Choi & Scacchi, 2001; Nissen, 1998; Scacchi, 2001; Scacchi & Nissen, 1998). Among the acquisition process pathologies we seek to identify are those where measured processes reveal sub-processes with *low effectiveness* (indicating high levels of iterative rework), *low efficiency* (indicating slow or bureaucratically cumbersome process steps that add marginal value to process completion), and *problematic sub-processes* (indicating underspecified process steps, steps that generate processing delays due to missing/or incorrect acquisition data, or inappropriate automated process enactment support). For example, current processes that assume long-term acquisition of monolithic software systems with proprietary components integrated within a closed architecture are likely not well-suited to address the challenges for acquiring secure OA systems that integrate software components from different online repositories.

We also place our acquisition workforce at a disadvantage if we do not empower them with the ability to measure, assess, and adaptively redesign their processes as technological advances like component-based OA systems are to be acquired. New software component technologies and software ecosystem niches (Scacchi & Alspaugh, 2012a) are also emerging, which necessitates new *continuous development processes* and new license management practices, and thus redesign and evolution of acquisition processes (Scacchi & Alspaugh, 2013a; Scacchi, Brown & Nies, 2012). These examples all point to new opportunities to redesign, evolve, or otherwise transform existing acquisition processes to better fit the challenges posed by the development, deployment, and support of secure, component-based OA systems. Finally, we can empower the acquisition workforce to realize continuously improved acquisition processes if we can provide them with the training and resources for modeling, analyzing, and redesigning their acquisition processes in ways that utilize Web-based automated process enactment systems, which also allow them to try out and walkthrough alternative process redesigns before committing to their use in daily operations,

***Design New Acquisition Processes***—Across the DoD community, there are many variations in practice for how to specify and model the architecture of a software-intensive system. Some practices focus attention primarily on identification of major components or abstract layers, while minimizing (or ignoring) attention to interfaces and interconnections, which are more challenging to identify and manage. However, the BBP initiative for OA systems points to the need for managing explicit i*nterface specifications* that identify and reinforce the use of *standard interfaces* (DAU, 2012). Without such interface and interconnection specifications, it is not possible to determine the scope or potential conflicts/matches between the IP licenses (and thus, TD rights) for the overall system architecture. In contrast, we have demonstrated in our prior research that component-based OA systems become tractable and evolvable from IP license management and security perspectives when the system architecture of components, connectors, and interfaces are explicitly modeled (Alspaugh & Scacchi, 2010; Scacchi & Alspaugh, 2011, 2012a, 2012b, 2013b). The use of standard interfaces further allows for simpler renderings of OA system structure, and thus simplifies license analysis. Further, once interfaces and interconnections become explicit, software component producers, system integrators, and system consumers can determine or negotiate which interfaces should be standardized in order to improve competition and affordability. These standards may then define acceptable data types, relationships between data types, data attribute value ranges, and exceptional data values in ways that are open, sharable, and reusable, as well as extensible when appropriate. Such improvements become possible by enabling an agile, adaptive ecosystem for software components of different size and capability relative to OA system product lines for different application domains (Reed et al., 2012; Scacchi & Alspaugh, 2012a, 2013b). Therefore, another important technique for streamlining the acquisition of secure, component-based OA systems, in line with BBP initiatives, is to provide the acquisition workforce with the resources and automated support to design and computationally enact new acquisition processes (i.e., explicitly modeled processes [Choi & Scacchi, 2001; Nissen, 1998; Scacchi, 2001; Scacchi & Nissen, 1998]), where the processes are open, agile, and adaptive. Such modeled processes may also then be shared, reused, continuously improved, and redistributed across the ecosystem of defense enterprises and program offices.

***Cost Management as a Process Design Element***—Part of the promise of the move to OA systems stems from their perceived potential to reduce acquisition life-cycle costs, improve competition, and improve defense affordability (DAU, 2012). But where and how are the associated cost factors or cost drivers for OA systems identified, tracked, and managed? After all, if we do not know where the cost factors are, or what activities, conditions, or events drive OA system acquisition costs, then we cannot effectively control such costs or make well-informed system capability–cost tradeoffs. For example, people who manage the acquisition of large-scale software systems within various defense enterprises are familiar with the many types of end-user license agreements for proprietary, closed source software systems (Anderson, 2012). In contrast, these people may not know how best to manage the acquisition of OA systems whose software components are jointly subject to different OSS or proprietary licenses.

The acquisition workforce has also learned in practice that software IP licenses are subject to change over time. However, one consequence is that long-lived or widely used software systems become more costly and much less amenable to technology substitution or vendor replacement, thereby reducing competition due to vendor lock-in. This works against defense affordability. In contrast, emerging online repositories offer different kinds of software component with different functional capabilities (described earlier), along with different IP licenses and end-user licenses (e.g., low cost, per user licenses). These repositories of software components represent a means for increased competition and affordability, but subject to different acquisition, development, or integration processes that are just coming to light. Accordingly, we believe that streamlining the acquisition process for secure, component-based OA systems requires that IP license cost obligations (e.g., license fees for end-user agreements) and license management regimes need to be incorporated into process measurement and assessment, process redesign and evolution, and design of new acquisition processes. This is also a subject for further acquisition research, but one offering practical near-term consequence.

## Conclusions

In this paper, we presented our current results from an ongoing investigation of how best to acquire secure open architecture (OA) software systems. These systems incorporate software product line (SPL) practices that include closed source proprietary software and open source software (OSS) components, where such components and overall system configurations are subject to different security requirements. The combination of SPLs and OSS components within secure OA systems represents a significant opportunity for reducing the acquisition costs of software-intensive systems by the DoD and other government agencies. Through our research efforts, we seek to make the acquisition of secure, component-based OA systems a simpler, more transparent, and more tractable process. Such a process must be easy to explicitly model, share, reuse, adapt, and streamline for different system application domains. Our goal was to identify ways and means for realizing cost reductions and improve acquisition workforce capabilities in ways that address Better Buying Power (BBP) initiatives associated with the move to OA systems and licenses (DAU, 2012).

In this paper, we identified different ways and means for how to streamline the acquisition process for secure OA software systems through a focus on doing more with limited resources. Central to our approach was our effort to identify and characterize new ways and means for acquisition process measurement and assessment, process redesign and evolution, design of new acquisition processes, and incorporation of cost factors and cost drivers as an element in new acquisition processes. Along the way, we paid particular attention to revealing how licensing practices for emerging online software component marketplaces can affect cost in ways that either hamper or better the buying power of acquisition programs. Consequently, we sought to identify possible next steps for new acquisition research that can more accelerate efforts to improve competition and defense affordability, as well as empower the acquisition workforce going forward, in ways aligned with BBP initiatives.

## Acknowledgements

The research described in this report was supported by grant #N00244-12-1-0004 from the Acquisition Research Program at the Naval Postgraduate School, Monterey, CA, and by grant #1256593 from the National Science Foundation. No endorsement, review, or approval is implied. This paper reflects the views and opinions of the authors, and does not represent the views or positions of any other group, enterprise, or government agency.

## References

Alspaugh, T. A, Scacchi, W., & Asuncion, H. (2010). Software licenses in context: The challenge of heterogeneously licensed systems, *Journal of the Association for Information Systems, 11*(11), 730–755.

Anderson, S. (2012). Software licensing—Smart spending in these changing times, *CHIPS: The Department of the Navy's Information Technology Magazine*, July–September, pp. 28–31.

Bergey, J., & Jones, L. (2010). Exploring acquisition strategies for adopting a software product line approach. In *Proceedings of the Seventh Annual Acquisition Research Symposium* (Vol. 1, pp. 111–122), Naval Postgraduate School, Monterey, CA.

Choi, S. J. & Scacchi, W. (2001). Modeling and simulating software acquisition process architectures, *Journal of Systems and Software, 59*, 343–354.

Defense Acquisition University (DAU). (2012). *Open systems architecture and technical data rights ... management approaches*. Retrieved from http://bbp.dau.mil/docs/Open%20Systems%20Architecture%20and%20Technical%20Data%20Rights%20.%20.%20.%20Management%20Approaches.pdf

Defense Information Systems Agency (DISA). (2012). *DoD open source and community source software development in Forge.mil* (SoftwareForge Document ID – doc26066doc26066). Retrieved from http://www.disa.mil/News/Conferences-and-Events/DISA- Mission-Partner-Conference-2012/~/media/Files/DISA/News/Conference/2012/DoD_Open_Source_Community_Forge.pdf

Department of Defense Open Systems Architecture (DoDOSA). (2011). *Contract guidebook for program managers* (Vol. 0.1). Retrieved from https://acc.dau.mil/OSAGuidebook

Garcia, P. (2010). Maritime C2 strategy: An innovative approach to system transformation. In *Proceedings of the 15th International Command & Control Research & Technology Symposium*, Paper 147, Santa Monica, CA.

Gizzi, N. (2011). Command and control rapid prototyping continuum (C2RPC) transition: Bridging the valley of death. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (Vol. 1), Naval Postgraduate School, Monterey, CA.

Guertin, N., & Clements, P. (2010). Comparing acquisition strategies: Open architecture versus product lines. In *Proceedings of the Seventh Annual Acquisition Research Symposium* (Vol. 1, pp. 78–90), Naval Postgraduate School, Monterey, CA.

Guertin, N., & Womble, B. (2012). Competition and the DoD marketplace. In *Proceedings of the Ninth Annual Acquisition Research Symposium* (Vol. 1, pp. 76–82), Naval Postgraduate School, Monterey, CA.

Hissam, S., Weinstock, C. B., & Bass, L. (2010). On open and collaborative software development in the DoD. In *Proceedings of the Seventh Annual Acquisition Research Symposium* (Vol. 1, pp. 219–235), Naval Postgraduate School, Monterey, CA.

Jones, L., & Bergey, J. (2011). An architecture-centric approach for acquiring software-reliant system. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (Vol. 1), Naval Postgraduate School, Monterey, CA.

Kenyon, H. (2012, October). DoD, Intel officials bullish on open source software; government-wide software foundation in the mix, *AOL Defense*.

Konary, A. (2009, October). *Software licensing and entitlement management: The next generation* (IDC White Paper). Retrieved from http://learn.flexerasoftware.com/content/ECM-WP-Software-Licensing-Entitlement-Management

Mactal, R., & Spruill, N. (2012). A framework for reuse in the DoN. In *Proceedings of the Ninth Annual Acquisition Research Symposium* (Vol. 1, pp. 149–164), Naval Postgraduate School, Monterey, CA.

Martin, G., & Lippold, A. (2011). Forge.mil: A case study for utilizing open source software inside of government. *Open Source Systems: Grounding Research*, Springer, pp. 334–337.

Naegle, B., & Petross, D. (2008). Software architecture: Managing design for achieving warfighter capability. In *Proceedings of the Fifth Annual Acquisition Research Symposium* (NPS-AM-07-104), Naval Postgraduate School, Monterey, CA.

Nissen, M. E. (1998). Redesigning reengineering through measurement-driven inference, *MIS Quarterly, 22*(4), 509–534.

Reed, H., Benito, P., Collens, J., & Stein, F. (2012, June). Supporting agile C2 with an agile and adaptive IT ecosystem. In the *17th International Command and Control Research and Technology Symposium* (ICCRTS), Paper-044, Fairfax, VA.

Scacchi, W. (2001). Redesigning contracted services procurement for Internet-based electronic commerce: A case study, *Journal of Information Technology and Management, 2*(3), 313–334.

Scacchi, W. (2007). Free/open source software development: Recent research results and methods, in M. Zelkowitz (Ed.), *Advances in Computers, 69*, 243–295.

Scacchi, W. (2009). Understanding requirements for open source software. In K. Lyytinen, P. Loucopoulos, J. Mylopoulos, & W. Robinson (Eds.), *Design requirements engineering: A ten-year perspective* (pp. 467–494 ), LNBIP 14, Springer Verlag.

Scacchi, W. (2010). The future of research in free/open source software development. In *Proceedings of the ACM Workshop on the Future of Software Engineering Research (FoSER)* (pp. 315–319), Santa Fe, NM.

Scacchi, W., & Alspaugh, T. (2008). Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In *Proceedings of the Fifth Annual Acquisition Research Symposium* (NPS-AM-08-036), Naval Postgraduate School, Monterey, CA.

Scacchi, W., & Alspaugh, T. (2011). Advances in the acquisition of secure systems based on open architectures. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (Vol. 1), Naval Postgraduate School, Monterey, CA.

Scacchi, W., & Alspaugh, T. (2012a). Understanding the role of licenses and evolution in open architecture software ecosystems, *Journal of Systems and Software*, *85*(7), 1479–1494.

Scacchi, W., & Alspaugh, T. (2012b). Addressing challenges in the acquisition of secure software systems with open architectures. In *Proceedings of the Ninth Annual Acquisition Research Symposium* (Vol. 1, pp. 165–184), Naval Postgraduate School, Monterey, CA.

Scacchi, W., & Alspaugh, T. (2013a). Processes in securing open architecture software systems. In *Proceedings of the 2013 International Conference on Software and System Processes*, San Francisco, CA.

Scacchi, W., & Alspaugh, T. (2013b). Challenges in the development and evolution of secure open architecture command and control systems. In *Proceedings of the 18th International Command and Control Research and Technology Symposium*, Paper-098, Alexandria, VA.

Scacchi, W., Brown, C., & Nies, K. (2012). Exploring the potential of virtual worlds for decentralized command and control. In *Proceedings of the 17th International Command and Control Research and Technology Symposium* (ICCRTS), Paper 096, Fairfax, VA.

Scacchi, W., & Noll, J. (1997). Process-driven intranets: Life cycle support for process reengineering, *IEEE Internet Computing, 1*(5), 42–49.

Northrop, L., & Clements, P. (2007). *A framework for software product line practice*, Version 5.0. Retrieved from http://www.sei.cmu.edu/productlines/frame_report/index.html

Womble, B., Schmidt, W., Arendt, M., & Fain, T. (2011). Delivering savings with open architecture and product lines. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (Vol. 1), Naval Postgraduate School, Monterey, CA.

# Processes in Securing Open Architecture Software Systems

## Abstract

Our goal is to identify and understand issues that arise in the development and evolution processes for securing open architecture (OA) software systems. OA software systems are those developed with a mix of closed source and open source software components that are configured via an explicit system architectural specification. Such a specification may serve as a reference model or product line model for a family of concurrently sustained OA system versions and variants. We employ a case study focusing on an OA software system whose security must be continually sustained throughout its ongoing development and evolution. We limit our focus to software processes surrounding the architectural design, continuous integration, release deployment, and evolution found in the OA system case study. We also focus on the role automated tools, software development support mechanisms, and development practices play in facilitating or constraining these processes through the case study. Our purpose is to identify issues that impinge on modeling (specification) and integration of these processes, and how automated tools mediate these processes, as emerging research problems areas for the software process research community. Finally, our study is informed by related research found in the prescriptive versus descriptive practice of these processes and tool usage in studies of conventional and open source software development projects.

**Categories and Subject Descriptors:** D.2.11 [Software Engineering]: Software Architectures

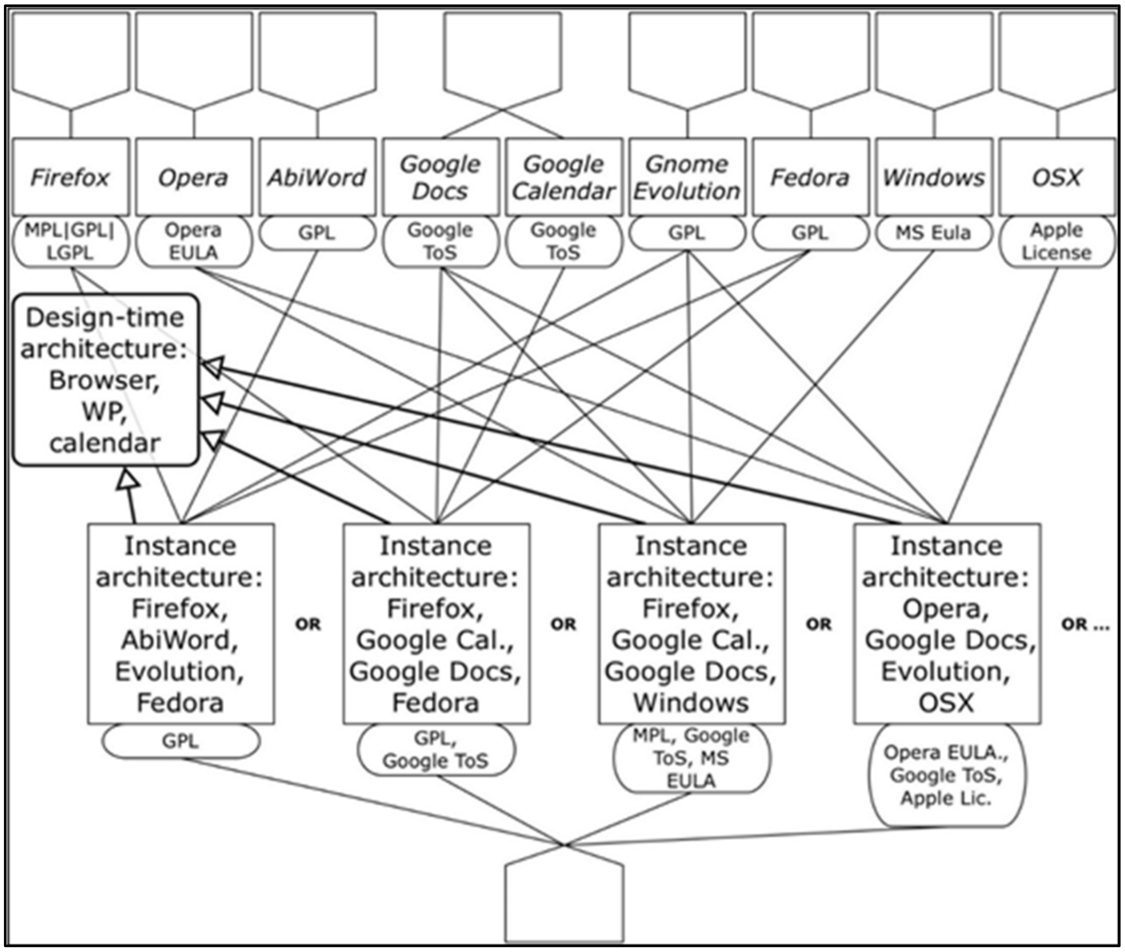
**General Terms:** Management, Security

**Keywords:** Open architecture, configuration, process modeling, process integration, continuous software development.

## Overview

Our goal is to identify and understand issues that arise in the development and evolution processes for securing open architecture (OA) software systems. OA software systems are those developed with a mix of closed source software (CSS) components with open APIs, and open source software (OSS) components, that are configured via an explicit system architectural specification. Such a specification may serve as a reference model or product line model for a family of concurrently sustained OA system versions and variants. We seek to research, develop, and refine new software process concepts, techniques, and tools for continuously assuring the security of large-scale OA software systems composed from software components that include proprietary CSS and non-proprietary and free OSS. In the U.S., federal government acquisition policy, as well as many leading enterprise IT centers, now encourages the use of CSS and OSS in the development, deployment, and evolution of complex, software-intensive OA systems.

In this paper, we employ a case study focusing on an OA software system whose security must be sustained throughout its ongoing development and evolution. We limit our focus to software processes surrounding the architectural design, continuous integration, release deployment, and evolution found in the OA system case study. To be clear, these processes focus on activities that construct and update configurations of software components, and are not the processes for developing the components themselves. The components involved in such OA systems have their own development life cycle, often within development projects that are independent or at arm’s length from the effort to develop and evolve an OA system composed from such components.



Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
MPL|GPL| LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
OR ...
GPL
GPL, Google ToS
MPL, Google ToS, MS EULA
Opera EULA., Google ToS, Apple Lic.

**Figure 1. A Software Ecosystem of Software Components that Can Be Configured into a Product Line Indicating Four Functionally Similar OA Systems**

In our case study, we examine a simple OA enterprise computing system that configures a Web browser (such as Firefox or Opera), word processor (such as AbiWord or Google Docs), email and calendar component (such as Gnome Evolution or Gmail), and operating system (such as RedHat Linux, RedHat Fedora with SELinux, Microsoft Windows, Apple OSX, or SEAndroid) in conjunction with file, mail, and Web servers (which may be on distributed network servers) in a loosely coupled manner. However, even this simple OA system that we study draws on an ecosystem of diverse software component providers, whose software products can be configured into alternative, functionally similar system configurations that conform to an OA software product family, as indicated in Figure 1. Such an OA system is also a core of more complex, mission-critical command and control systems (Gizzi, 2011; Scacchi, Brown, & Nies, 2012). Additionally, such a system can also be built and deployed for use on a mobile computing platform like a tablet or smartphone. Finally, our OA system can be encapsulated within security capability and enforcement mechanisms (e.g., SELinux capabilities, virtual machine hypervisors) in order to secure the OA system (Defense Information Systems Agency [DISA], 2012; Smalley, 2012; US-CERT, 2011; Xen Hypervisor Project, 2013).

We also use the case study to focus on the role automated tools, software development support mechanisms, and development practices play in facilitating or constraining OA software processes. Our purpose is to identify issues impinging on modeling (specifying) and integrating these processes and explore how automated tools mediate these processes, as emerging research problems areas for the software process research community. We also discuss how such issues affect practical simulation and analysis of these processes.

In the remaining sections of this chapter, we first examine related research found in the prescriptive versus descriptive practice of software processes for architectural design, continuous integration, release deployment, and evolution. Next is our case study, describing an OA enterprise computing system that must remain continually secure as it evolves; we use this to help identify issues arising in the specification and integration of the four software processes when the goal of the overall process effort is to continually secure an OA system. We present examples throughout this case study. We then investigate the software process modeling and process integration issues that were observed in this study, as well as how they further constrain efforts to simulate or computationally analyze such processes, and conclude the paper.

## Related Research and Development Efforts

We choose to focus on the processes from architectural design, continuous integration, and release deployment to software evolution for OA systems. Such systems incorporate both CSS and OSS components. In particular, our interest is to examine how these processes enable or constrain how to produce a secure OA system. In particular, we recognized that processes for software architecture design and software evolution (Madhavji, Fernandez-Ramil, & Perry, 2006) have received prior attention in the software process community, but continuous integration and release deployment have received much less attention. Similarly, relatively little is known about how design processes enable and constrain continuous integration and delivery, and how they in turn facilitate or constrain software evolution. Such an undertaking needs to go beyond prior efforts to specify and identify issues that may arise in processes for the development of component-based software systems (Crnkovic, Chaudron, & Larsson, 2006; Qureshi & Hussain, 2008). Earlier process studies like these do not address, for example, how new development technologies such as continuous integration systems mediate development processes for component-based systems. They also do not identify continuous integration, or software release delivery and installation, as salient development processes for component-based software systems. This may be so as continuous integration and release management are relatively new software development processes, and such processes seem to be visibly practiced in large OSS development projects. Finally, these earlier studies offer little insight as to how functional or non-functional requirements for securing an OA system mediate its software development and evolution processes. But we do know some things about these processes from related efforts, especially for continuous integration.

Continuous integration (CI) systems support automated processes for building, testing, and packaging a software system for release ("Continuous Integration," 2012; Duvall, Matyas, & Glover, 2007; Fowler, 2000;). Without a CI system, developers must build, test, and integrate their software (component) products using hand-crafted scripts, and it is common for such scripts to have to rely on idiosyncratic dependencies on tool chains and libraries versions for each deployment platform targeted (e.g., Hypertable, 2013). In contrast, CI systems incorporate the capabilities of software build systems (Smith, 2011) that may invoke sequential, distributed, or parallel builds across multiple build servers (cf. ThoughtWorks CI Feature Matrix, 2012) to produce singular builds (e.g., "nightly builds"), continuously updated agile development builds (Fowler, 2000), or diverse, functionally equivalent executable variants (Jackson et al., 2011). The build systems access and update software code (version control) repositories via process automation scripts. CI sub-processes take as input directories and folders of source code files and produce software component executables. The executables may also be organized as a structured collection (an information architecture) of binary files, static data value and parameter setting files packaged in interlinked directories, constituting releases for deployment. Continuous delivery (CD) further extends CI to support automated release management and the creation of automated deployment tools such as "installation wizards" to be used by system administrators or end-users (Humble & Farley, 2010). For the remainder of our paper, we use the abbreviations CI and CD to refer to these sets of automatable software development processes.

As Fowler (2000) observed about the need for continuous integration as an enabling mechanism for agile development, "the key is to automate absolutely everything and run the process so often that integration errors are found quickly. As a result everyone is more prepared to change things when they need to, because they know that *if they do cause an integration error, it's easy to find and fix*" (emphasis added). CI processes can therefore be viewed with the assumption that errors resulting from process automation are normal, expected, and not necessarily easily anticipated. But why do these errors occur at all, and why do we need to run the process often in order to identify and resolve integration problems? We need to make closer, systematic observations to determine why or how these errors occur, so that we can advance our process engineering knowledge, as well as enable practical process improvement. A case study can serve as a starting point for this, and this is our strategy.

Automated CI systems comprise composed environments of software tools, or sets of loosely coupled tools together by automated process invocation scripts that guide and constrain their use. Often these tools are independently developed and evolved. For example, a CI system like Hudson (Hudson-ci, 2011) includes source code build tools like Ant or Maven, an issue tracking (or bug reporting) tool like Bugzilla (Jensen & Scacchi, 2005) or Jira, and a software revision control browser and search engine like FishEye or ViewVC for viewing the contents of software revision control code repositories like CVS or Subversion. All of these tools happen to be OSS associated with active OSS development projects, so these tools are subject to ongoing development and evolution that improve their capabilities and add/remove functionality. Other CI systems may use different tools or locally developed capabilities in place of external OSS tools such as these. Consequently, this implies the process steps enacted by a CI system will vary (and evolve) depending on the choice of CI system, and on the external tools or locally embedded software functionality that a particular CI system uses. Whether such CI process steps are equivalent, similar, or incongruent across CI systems thus remains an open issue. But it is an issue that must be resolved when transitioning from one CI system, or CI system version, to another. However, current CI systems do not appear to address this, nor do they identify it as a concern in their recommended best practices (cf. Hudson-ci, 2011; ThoughtWorks CI Feature Matrix, 2012). Similarly, when we add the need to address the CI and CD of secure OA systems, we quickly finds gaps in the best practices that point to shortfalls either on the CI/CD process support side, the security capability side (US-CERT, 2011), or their interdependencies.

Automated CI systems are continuously being improved or supplanted (Jenkins, 2013; Krill, 2011) and different CI systems offer different features, functional capabilities, and depend on different software tools (ThoughtWorks CI Feature Matrix, 2012). The same can be said for CD/release deployment systems, especially with regard to ongoing advances and refinement of software packagers, file distribution and mirror (copy server) synchronization, installers, and uninstallers (Humble & Farley, 2010). So, from a software process specification or modeling viewpoint, there are many distinct CI process instance types, and no single abstract CI or release deployment process prescription to follow and tailor to local development organization needs. CI and CD process enactment must therefore rely on manual best practices in addition to tool-based automation, and these practices are specific to each CI system and the tools therein (Hudson-ci, 2011). CI and release management system-based process automation thus are both ad hoc and idiosyncratic, rather than easily standardized or generalized, and yet are a widespread software engineering process and practice used to produce thousands of software components (e.g., smartphone or tablet apps).

Software delivery and deployment suffer similar kinds of process automation pathologies (e.g., IBM Software Group, 2007), to the extent that a key advantage of automation is now thought to be finding or process enactment errors, mistakes, or other articulation problems (Mi & Scacchi, 1991) by running the enactment more quickly. Software deployment errors, such as releasing and installing a premature system release candidate into production operations can have devastating technical or economic consequences, as was demonstrated by the experience of Knight Capital in the summer of 2012 (Dignan, 2012). How to provide automated tools and practical techniques that provide more robust acceptance and compliance checking prior to a new system version being installed prior to going live in operation, seems to be an underspecified process enactment problem. Adding robust diversity mechanisms and capabilities for dramatically improving OA system security (Gorlick, Strasser, & Taylor, 2012; Jackson et al., 2011; Scacchi & Alspaugh, 2013) remains an open question for further study. Once again, a case study can serve as a starting point for examining such issues and concerns, and this is our strategy.

We see that part of the process challenge is how to understand and specify software processes that must interface with emerging CI and CD systems. These CI systems entail different kinds with different build, package, and release deployment process automation capabilities, or that produce integrated systems that operate on different platforms (ThoughtWorks CI Feature Matrix, 2012). To us, this raises concerns for process specification—determining what aspects of a software process are pertinent for modeling and simulation, as well as contributory to improving process effectiveness (Nichols, Kirwan, & Andelfinger, 2011), and process integration—integrating modeled process specifications with diverse automated process enactment mechanisms (Mi & Scacchi, 1992). It also raises issues for integration across multiple process representations that are supported by independently developed, heterogeneous process enactment mechanisms (Garg, Mi, Pham, Scacchi, & Thunquest, 1994).

## Case Study: A Secure OA Enterprise/C2 System

We utilize a case study to explore and identify software process issues that arise while producing a secure enterprise computing software system. Such a system is produced using existing software applications as components, composing and configuring them to realize the overall system. The processes we examine are not those that develop such software applications, but rather those that use them as components of the system. However, this choice still highlights how the ongoing, independent development and evolution of the components motivates new versions/variants of the overall OA system. In this regard, software component evolution is a driving force that impinges on the development and evolution of OA systems incorporating such components.

Another aspect of our study is to recognize some software processes, like architectural design and software evolution, as having limited automated enactment, while others such as continuous integration and release management are potentially fully automated. This is not to say that no tools are involved in design or evolution, far from it. Rather, what is of interest is that software production and system integration organizations employ a flow of software processes that employ both fully and partially automated enactment. Assuming a world where all software processes are fully automated may be another challenge, but it is not one that is of practical use or consequence at this time. Our study thus addresses software process challenges that are both reflective of understanding of emerging software process research issues, and also may have practical application today and beyond. As such, we turn to our case study to elaborate the software processes of interest and to the issues they raise for software process research.

## Architectural Design Process

The process for designing the configuration of an OA system at the component level is our focus here. We start by noting that we assume no pre-existing process model or standard for such a process, nor do we propose to provide such a prescriptive process. As a review of the architectures of dozens of OSS systems (Brown & Wilson, 2012) makes clear, there is no common prescriptive process or preferred set of tools, nor is there a notational scheme for the architectural design of open software systems. Instead, we describe aspects of a design process we developed, practiced, and adapted that is supported in part with automated design tools. One of our goals with this process was to help identify situations, and practical non-functional requirements, that arise with an OA design process that constrains, and is constrained by, the other three downstream software processes in our study.

We have used an OA tailored version of the UCI ArchStudio4 architecture design system (oAS4) as a locally developed plug-in to the Eclipse IDE to realize a partially automated system for architectural design activities (Alspaugh, Asuncion, & Scacchi, 2012, 2013). oAS4 allows us to visually model the architectural configuration of software components, component interfaces, and component connectors as OA system elements. oAS4 also produces output in an architectural description language (ADL) as a persistent artifact for external analysis, or for potential integration with CI systems with further processing (e.g., binding component classes to their build-time instances). We further focus our architectural design activities to produce an abstract system architecture that serves to denote a product line model of a family of alternative system configurations composed from functionally similar components or component versions (Scacchi & Alspaugh, 2012). oAS4 can thus support our experimental studies in OA system design and design evolution across families of alternative system configurations (cf. an earlier approach to such problems at Narayanaswamy & Scacchi, 1987).

We annotate our OA system designs within oAS4 using formal constraint expressions on components interfaces, such as intellectual property (IP) license obligations and rights (Alspaugh et al., 2012, 2013). Security policy constraints for components, configured sub-systems, or an overall system are expressed and analyzed in a similar manner (Scacchi & Alspaugh, 2013). The ability to model and automatically analyze such obligations and rights is needed at build-time and release deployment-time. Automated analysis mechanisms then allow us to determine whether the specified component interconnections entail matches or conflicts in component–component license alignments (Alspaugh et al., 2012, 2013). However, we have also observed that design-time actions must accommodate build-time and deployment-time element bindings, as well as accommodate the evolution of licenses, policies, and system element versions  (Scacchi & Alspaugh, 2012). For example, when conflicts are found between the licenses of interconnected build-time component selections, we can then reconfigure our OA system design to eliminate the conflicts, to constrain the selection of components at build-time (within CI) to those whose licenses will match or not conflict, or to wrap or shim a component with an abstraction layer that does not transfer IP license obligations.

Design of OA systems also raises issues for how to how best to secure the designed system architecture (US-CERT, 2011). Among the recommended practices for designing secure system architectures are providing capability-based user/developer access control that effectively limits access to input and output data, internal program code representations (e.g., memory address and system name spaces), persistent data storage, and to exposed I/O transaction processing interfaces. One increasingly common approach is to provide encapsulation mechanisms like virtual machines for software components or (sub-)system configurations, along with encrypted inter-component data/control flow connectors (e.g., HTTPS/SSL data communication protocols). Of these, passively secure connectors for networked components are widely available, while dynamically secured connectors are a recent advance (Gorlick et al., 2012). In our case, we choose to incorporate virtual machines to encapsulate our OA system, and we ignore alternative security protection schemes for simplicity. However, we recognized that even a seemingly simple decision like this still requires analyzing trade-offs about whether to encapsulate the entire system as a single virtual machine (relatively easy to address during deployment, though requiring deployment and installation of virtual machine software [e.g., Xen Hypervisor Project, 2013] on the target deployment computers) or to encapsulate each different component within its own virtual machine that would then be interconnected using secure connectors (more challenging to address for deployment, but offering a more resilient OA system security [Scacchi & Alspaugh, 2013]). We decided to design something in-between these two extremes, by taking into account where different components might be hosted within a networked, multi-server platform environment. What our OA system design process produced is an abstract architectural configuration of component types (each attributed with IP license constraints—not shown but described elsewhere; Alspaugh et al., 2012, 2013; Scacchi & Alspaugh, 2013), a minimal component interconnection scheme, and what we call a hybrid virtual machine confinement scheme, as shown in Figure 2.

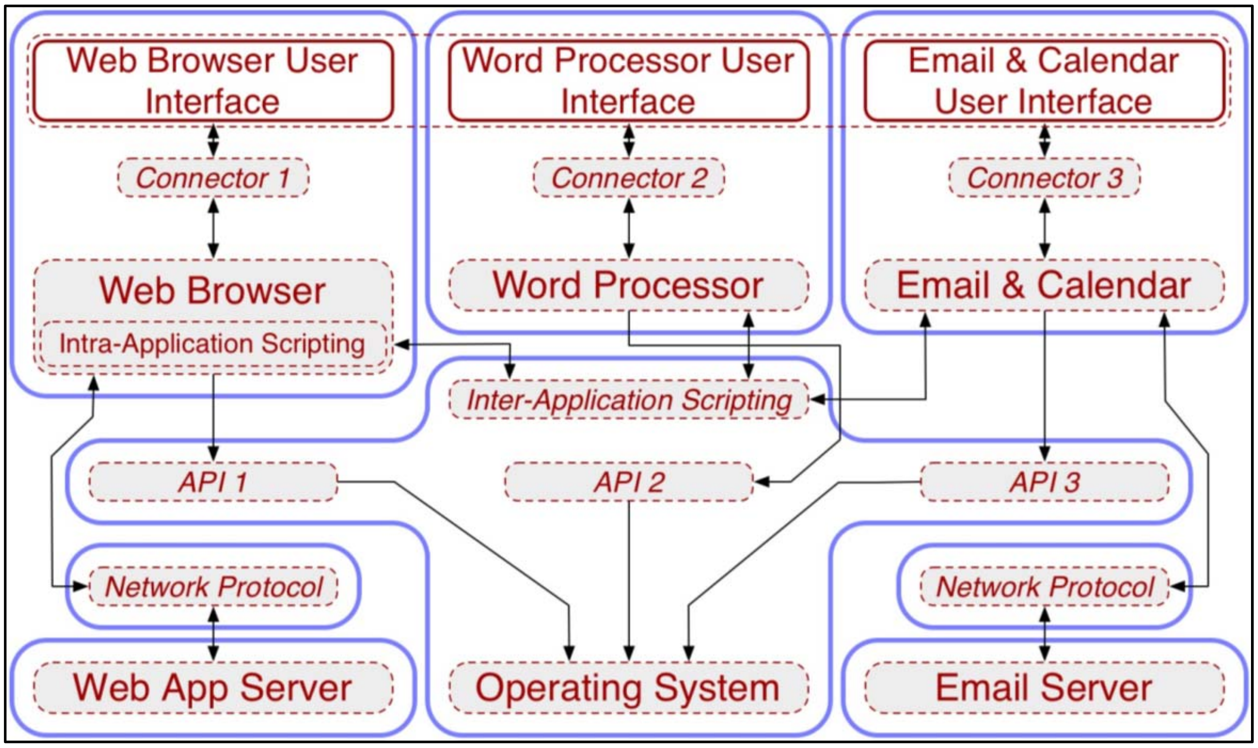


**Figure 2. Design Configuration of a Secure OA Enterprise/C2 System, Shown With Security Encapsulation Layout**

Given that we have so far only examined the architectural design process, we note that we are already beginning to see that we need to anticipate non-functional requirements for the other downstream software processes that follow, particularly in the form of process enactment directives or constraints. We also begin to anticipate whether such information can be automatically propagated into the process automation tools used in these downstream processes.

## Continuous Integration Process

In our study, one of the first activities in moving from architectural design to continuous integration is to identify specific software component versions that can be instantiated within the current architectural configuration (Figure 2). While at first it might seem that this is a simple task, we have found that component and version selection are subject to the obligations and rights stipulated with a component's associated IP license (Alspaugh et al., 2012). For example, common architectural design languages do not specify annotations for IP licenses, so as noted above, we extended our ADL within the oAS4 with IP obligation and right constraints (Alspaugh et al., 2012, 2013). This meant we could now analyze whether or how IP obligations and rights for each component–component interconnection match, conflict, or propagate. For example, reciprocal licenses like GPL can propagate their IP regime by design, though some enterprises seek to avoid this. By conceptually filling in selected component licenses, we can tell, prior to integration, whether the resulting release candidate may suffer from licensing problems or not. When conflicts or mis-matches are discovered, again prior to further build-time process actions, alternative components with the similar functional capabilities and interfaces but different licenses may be substituted. Alternatively, the architectural configuration can be modified, for example, wrapping a component in a way that mitigates license conflicts (e.g., replacing a direct API–API interconnection which propagates license restrictions with a networked data communications link, as few licenses propagate IP across network connections).

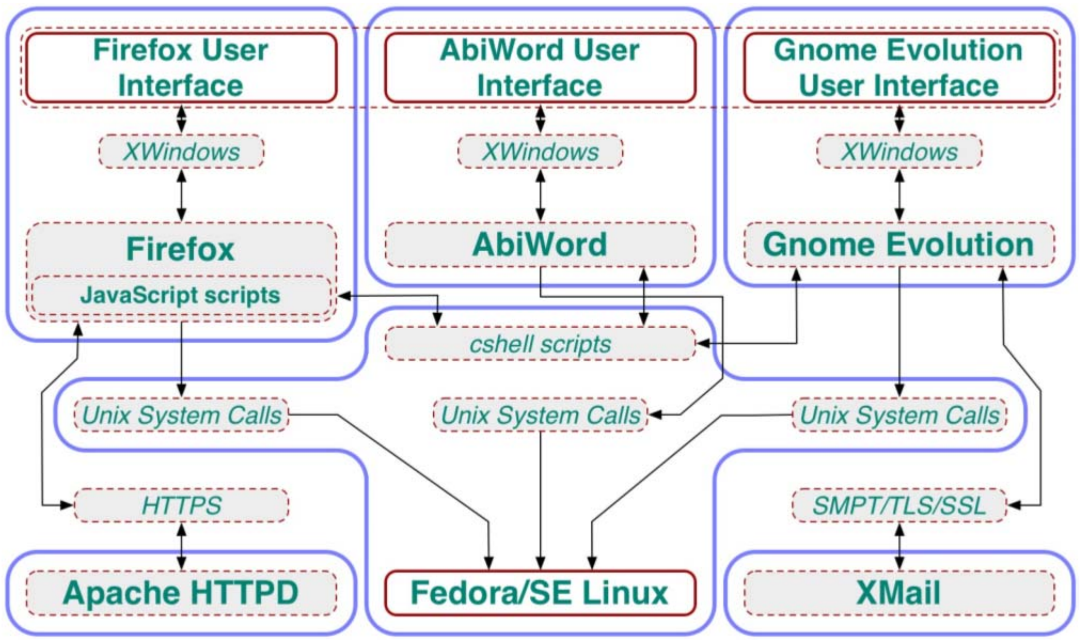


**Figure 3. A Build-time Configuration of a Secure OA Enterprise/C2 Computing System**

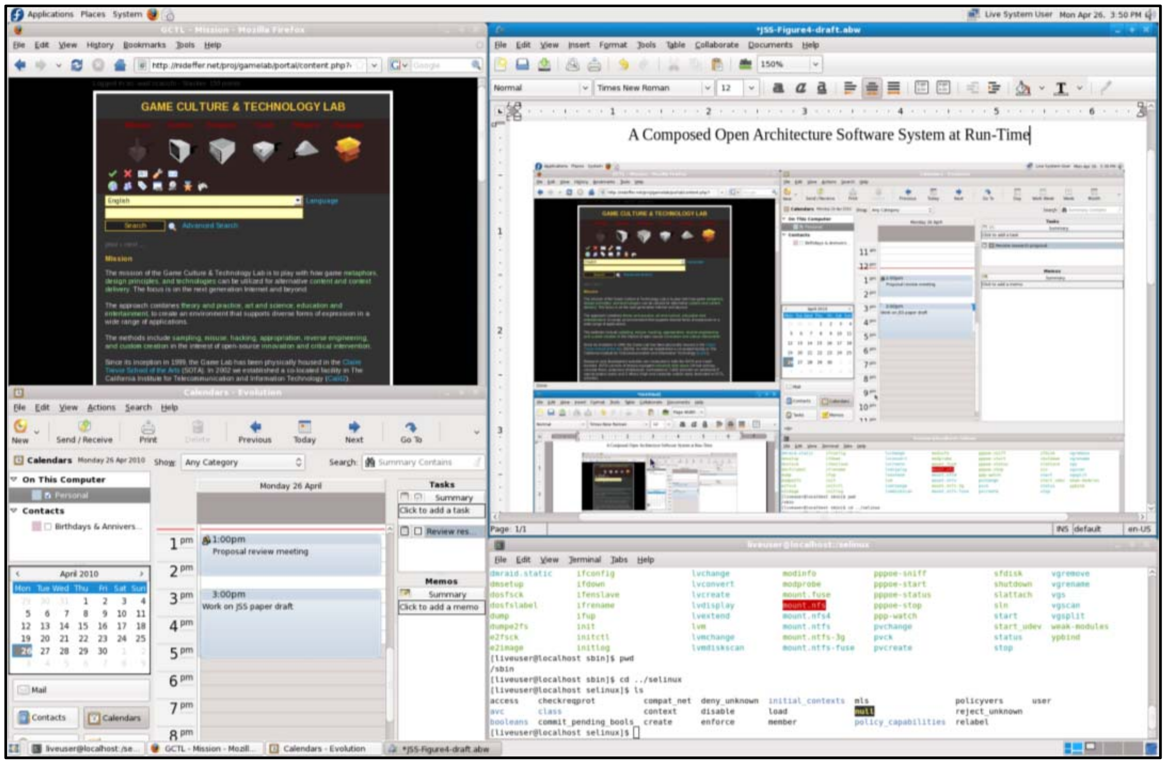
What we end up with from our build sub-process is a concrete OA system configuration with a specific selection of software components specified using oAS4, whose output is intended for a manual build system or for entry into an automated CI system. A concrete configuration is seen in Figure 3. So our build sub-process can now instantiate components into a reusable OA software product line design, as we can determine families of component version instances that can be substituted within the OA system. For example, the Firefox Web browser may be replaced by Google Chrome in this configuration, because both are under permissive OSS licenses. However, a license match/conflict assessment would be required before replacing Firefox with Microsoft Internet Explorer (IE) or Opera, each of which is under a proprietary license. But in the abstract and concrete architectural configuration we have, we could substitute a Linux-based Opera browser without issue, but not IE, unless we add a library wrapper such as Wine (WineHQ, 2013), in order to run IE on Fedora Linux.

So far, so good. But now we must consider how to transfer this component selection specification into the build system arises. An ideal solution might involve an automated hand-off. However, the specifics of such a hand-off will vary depending on the build system and the CI system we select. A more general solution would likely require (or benefit from) another abstraction layer for integration between the architectural design and build/CI process enactment mechanisms, which is an already recognized problem with a demonstrable solution (cf. Garg et al., 1994). We see that software process research may demonstrate solutions to messy process integration issues, but integration of process flows across tool-specific process enactment representations and automated mechanisms remains a lingering, practical problem that is not yet addressed by current CI or CD systems.

A similar problem arises when we consider how to secure the concrete OA system configuration. For example, we can choose to include secure data communication connectors (e.g., secure protocols like HTTPS and TLS/SSL) in our configuration, but such capabilities are not instantiated at build-time. Instead, they depend on mechanisms and data (e.g., certificates) that are accessed at run-time once an integrated system release candidate is available. An OA system, or OA system components, can also be secured using virtual machine hypervisors (Xen Hypervisor Project, 2013) that confine and isolate deployed system/component within a virtual machine run-time environment. In addition, it should be possible to specify operating system access control and type enforcement capabilities (e.g., using SELinux libraries on Fedora), but again, these are not available for use until there is a deployable integrated system release candidate. Thus, these forms of security are most likely invisible to current CI systems and must be addressed through other means.

## Release Deployment Process

The software system you release and deploy depends on what (and how) you build and package for release and installation. For example, in our enterprise system, we want our software integration process to produce a run-time version of our designed software configuration for our target platform (e.g., local personal computer). Figure 4 displays a run-time instantiation in operation, based on the build-time configuration in Figure 3, hosted on a Fedora Linux operating system that utilizes the SELinux library to set access control and run-time capabilities for files and programs.


Applications Places System
Live System User Mon Apr 26, 3:50 PM
GCTL - Mission - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://nideffer.net/proj/gamelab/portal/content.php?
GAME CULTURE & TECHNOLOGY LAB
English
Language
Search
Advanced Search
Mission
*JSS-Figure4-draft.abw
File Edit View Insert Format Tools Table Collaborate Documents Help
150%
Normal
Times New Roman
12
Page: 1/1
INS default
en-US
Calendars - Evolution
File Edit View Actions Search Help
New
Send / Receive
Print
Delete
Previous
Today
Next
Go To
Calendars Monday 26 Apr 2010
Show: Any Category
Search: Summary Contains
On This Computer
Personal
Contacts
Birthdays & Anniversaries
Monday 26 April
Tasks
Summary
Click to add a task
Review res...
Memos
Summary
Click to add a memo
1:00pm Proposal review meeting
3:00pm Work on JSS paper draft
April 2010
Mail
Contacts
Calendars
liveuser@localhost:/selinux
File Edit View Terminal Tabs Help
liveuser@localhost:/se...
GCTL - Mission - Mozill...
Calendars - Evolution
*JSS-Figure4-draft.abw

A Composed Open Architecture Software System at Run-Time

The mission of the Game Culture & Technology Lab is to play with how game metaphors, design principles, and technologies can be utilized for alternative content and context delivery. The focus is on the next generation Internet and beyond.

The approach combines theory and practice, art and science, education and entertainment, to create an environment that supports diverse forms of expression in a wide range of applications.

The methods include sampling, misuse, hacking, appropriation, reverse engineering, and custom creation in the interest of open-source innovation and critical intervention.

Since its inception in 1999, the Game Lab has been physically housed in the Claire Trevor School of the Arts (SOTA). In 2002 we established a co-located facility in The California Institute for Telecommunication and Information Technology (Calit2).

```
dmraid.static    ifconfig     lvchange      modinfo           pppoe-sniff    sfdisk       vgremove
dmsetup          ifdown       lvconvert     modprobe          pppoe-start    shutdown     vgrename
dosfsck          ifenslave    lvcreate      mount.fuse        pppoe-status   slattach     vgs
dosfslabel       ifrename     lvdisplay     mount.nfs         pppoe-stop     sln          vgscan
dump             ifup         lvextend      mount.nfs4        ppp-watch      start        vgsplit
dumpe2fs         init         lvm           mount.ntfs        pvchange       start_udev   weak-modules
e2fsck           initctl      lvmchange     mount.ntfs-3g     pvck           status       ypbind
e2image          initlog      lvmdiskscan   mount.ntfs-fuse   pvcreate       stop
[liveuser@localhost sbin]$ pwd
/sbin
[liveuser@localhost sbin]$ cd ../selinux
[liveuser@localhost selinux]$ ls
access    checkreqprot          compat_net  deny_unknown  initial_contexts  mls                    policyvers      user
avc       class                 context     disable       load              null                   reject_unknown
booleans  commit_pending_bools  create      enforce       member            policy_capabilities    relabel
[liveuser@localhost selinux]$ 
```

**Figure 4. A Screenshot View of a Deployed Release Configuration of Our OA Enterprise/C2 Computing System**

However, what we build and what we release may not be the same, though they need to be functionally equivalent. For example, when we select one or more CSS components (an already compiled and integrated executable binary image) with a common restrictive IP license (i.e., one that prohibits copying or redistribution) for inclusion in our build-time architectural configuration, during the build process, we must link it as an executable binary for inclusion in a release candidate for deployment (or deployment testing; cf. Jensen & Scacchi, 2005) on a local computer. Such inclusion is a prerequisite for overall integrated system testing processes required by CI. However, we cannot distribute such a release candidate to others, as it is common for CSS to not allow duplication or distribution of licensed copies of software binaries. Instead, we need to specify and configure a deployment-platform specific automated software installation mechanism (e.g., installation wizard) that needs to search for and find a local licensed copy of the CSS executable binary, and link it to the result of the build sub-process that provides a run-time linkage mechanism in expectation. A similar effort is needed to enable user acceptance testing or certification testing on their local platform. These release deployment process steps can be accomplished with some effort, but this effort could also be anticipated at design-time or build-time, when developers make their selection for which component instances to include in the system build.

Automated software installation is an increasingly common expectation. Software installers run automated process enactment scripts crafted by developers. Once again, the process being enacted is not explicitly specified, nor is it separate from the internally coded software utility's action invocation scripts. This means that it is not surprising to discover errors that arise during installation but are not easily anticipated without extensive prior experience in working with the installer on known target platforms. For example, an informal aid from IBM for guiding system administrators who enact software installation processes (IBM Software Group, 2007) notes installation problems like (a) insufficient free space on disk storage prior to or during software executable installation; (b) software installations across a network that are "hung" or stuck due to lack of robust installation protocols that can time-out (abort) and/or re-initiate then re-validate process script commands already invoked; (c) installations that fail due to underspecified all/nothing installation transactions (cf. Gray, 1981) that do not completely update the information architecture of a multi-part software configuration (e.g., program registry update and reversible roll-back to prior registry; and/or setup of user configuration files); (d) failure to include a software uninstaller (or uninstallation process scripts) that allows conditional roll-back to previously installed software versions to be retrieved and activated; or (e) file/directory name collisions that arise at build-time versus deployment-time.

Our observation is that if there is a sufficiently detailed, informing process specification or model for how best to install a software release, it is well hidden. We all rely on the correct operation and outcome on software installation processes on our networked personal computers and wireless mobile devices, but such processes often are problematic or fail. This situation is not inevitable, but it is widespread. There is a missed opportunity to improve the quality of release deployment process outcomes by some means other than the costly software installation trial and error learning experiences that afflict software release deployment personnel and system administrators. We should be able to do much better than this. The provision of explicit software installation process models that can guide the targeting of different deployment platforms in specific organizations or for remote users begs for research and development attention.

## Evolution Process

An OA system can evolve by a number of distinct mechanisms or process enactment pathways, some of which are common to all systems, but others of which arise only in OA systems or where components in a single system are heterogeneously licensed (Alspaugh et al., 2013). Figure 5 provides a summary of some of the various paths, further explained below.

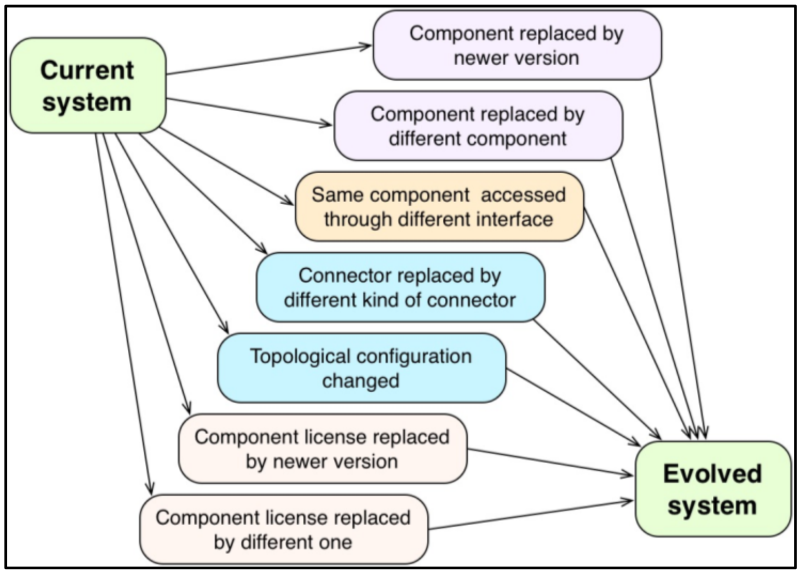

Current system
Component replaced by newer version
Component replaced by different component
Same component accessed through different interface
Connector replaced by different kind of connector
Topological configuration changed
Component license replaced by newer version
Component license replaced by different one
Evolved system

**Figure 5. A Variety of Paths and Activities Accounting for the Evolution of OA Systems**
(Scacchi & Alspaugh 2012)

***Component Version Evolution***—One or more components can evolve, altering the overall system’s characteristics. An example is upgrading the Firefox Web browser from version 17.0 to 17.1. Such minor version changes generally have no effect on system architecture. However, many large enterprises choose to sustain their software systems by relying on long-term support (LTS) versions of software components, rather than automatically updating to each release from software component producers. Instead, LTS components are replaced with new versions only over long time frames, where the new LTS version for installation may skip many intervening release versions. Such enterprises rely on local patches and workarounds between the LTS versions, under the belief that such an approach provides increased system stability and allows more comprehensive regression testing prior to deployment. But in these days of relentless attacks on system security, using LTS components entails locally sustaining system component or configuration versions with known vulnerabilities, often without code repositories that match those employed for CI. The vulnerabilities must then be defended using separate, orthogonal system security mechanisms, such as virtual machines or hypervisors from VMWare or Xen (Xen Hypervisor Project, 2013). Once again, we can do better than this through the use of explicit process specifications that model and provide process integration support across CI and CD systems, along with code repositories.

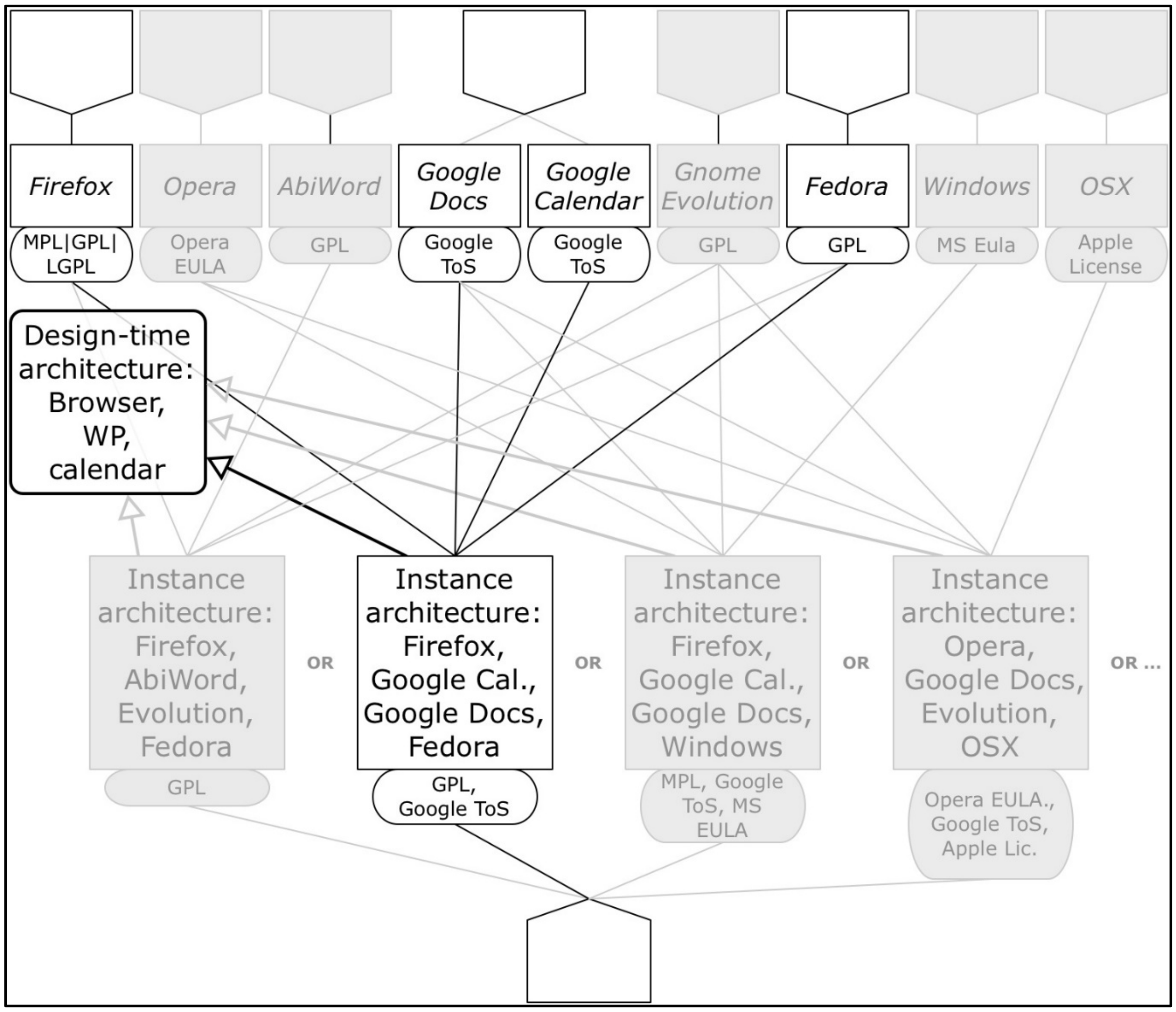
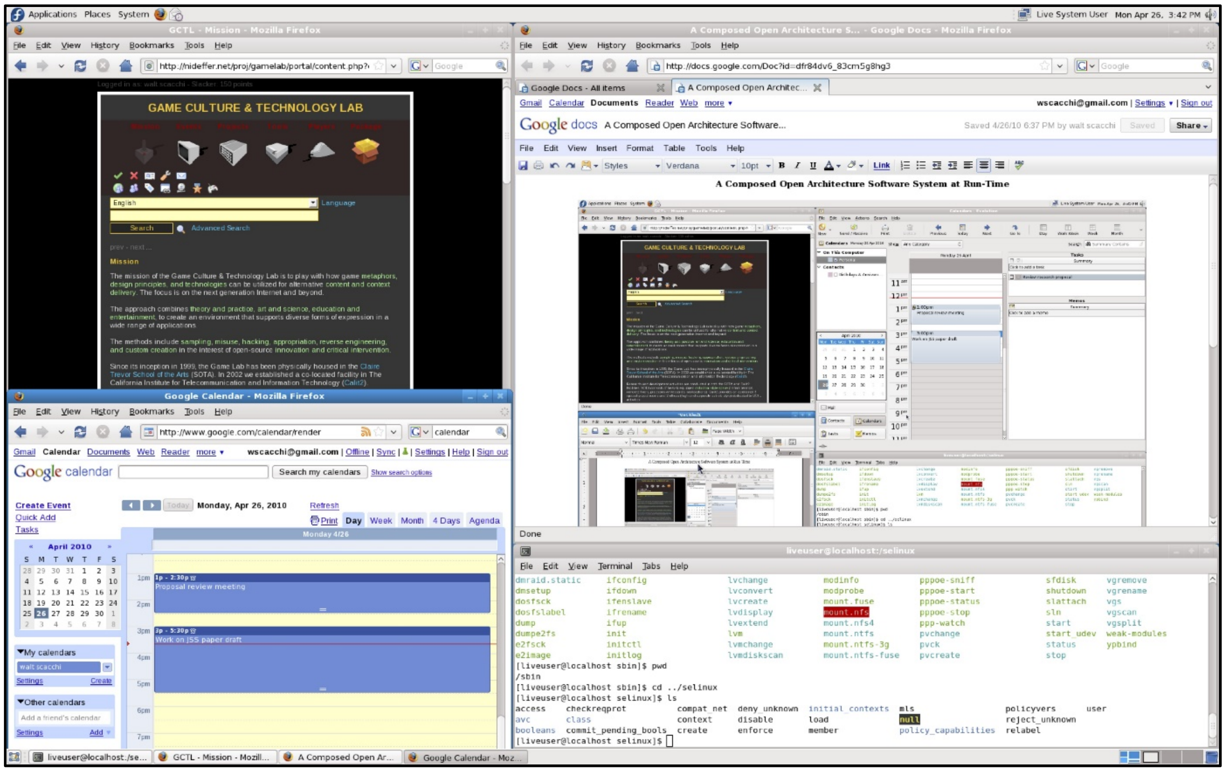

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
OR ...
GPL
GPL, Google ToS
MPL, Google ToS, MS EULA
Opera EULA., Google ToS, Apple Lic.

**Figure 6. An Alternative OA System Configuration Resulting From Replacement of Selected Components Shown in Figure 4 During System Evolution**
(Scacchi & Alspaugh, 2012)

***Architectural Configuration Evolution***—The OA can evolve by changing the kinds of connectors between components, rearranging connectors in a different configuration, or changing the interface through which a connector accesses a component, altering the system characteristics. Revising or refactoring the configuration in which a component is connected can change how its license affects the rights and obligations for the overall system. An example is the replacement of components for word processing, calendaring, and email with Web browser–based services such as Google Docs, Google Calendar, and Google Mail. The replacement would eliminate the legacy components and relocate the desired application functionality; it would operate remotely, but interact from within the local Web browser component. The resulting system architecture might be considered simpler and easier to maintain, but is also less open and now subject to a proprietary Terms of Service license. Ongoing evolution and support of this subsystem is now beyond the control and responsibility of the local system developers. System consumer preferences for one kind of license over another, and the consequences of subsequent participation in a different OA system evolution regime, may thus determine whether such an alternative system architecture is desirable or not. Figures 6 and 7 show examples of such evolutions in architectural configuration at release deployment time. These figures can be compared to the system deployment in Figure 4, but now where the build-time architecture now reconfigures the word processor, email, and calendaring into the single Web browser component, thus refactoring the build-time and release deployment-time system configurations, while remaining within the design-time product family indicated in Figures 2 and 6.


GCTL - Mission - Mozilla Firefox
GAME CULTURE & TECHNOLOGY LAB
A Composed Open Architecture S... - Google Docs - Mozilla Firefox
A Composed Open Architecture Software System at Run-Time
Google Calendar - Mozilla Firefox
Monday, Apr 26, 2010
liveuser@localhost:/selinux

Mission

The mission of the Game Culture & Technology Lab is to play with how game metaphors, design principles, and technologies can be utilized for alternative content and context delivery. The focus is on the next generation Internet and beyond.

The approach combines theory and practice, art and science, education and entertainment, to create an environment that supports diverse forms of expression in a wide range of applications.

The methods include sampling, misuse, hacking, appropriation, reverse engineering, and custom creation in the interest of open-source innovation and critical intervention.

Since its inception in 1999, the Game Lab has been physically housed in the Claire Trevor School of the Arts (SOTA). In 2002 we established a co-located facility in The California Institute for Telecommunication and Information Technology (Calit2).

1p - 2:30p Proposal review meeting

3p - 5:30p Work on JSS paper draft

```
dmraid.static   ifconfig     lvchange     modinfo          pppoe-sniff    sfdisk      vgremove
dmsetup         ifdown       lvconvert    modprobe         pppoe-start    shutdown    vgrename
dosfsck         ifenslave    lvcreate     mount.fuse       pppoe-status   slattach    vgs
dosfslabel      ifrename     lvdisplay    mount.nfs        pppoe-stop     sln         vgscan
dump            ifup         lvextend     mount.nfs4       ppp-watch      start       vgsplit
dumpe2fs        init         lvm          mount.ntfs       pvchange       start_udev  weak-modules
e2fsck          initctl      lvmchange    mount.ntfs-3g    pvck           status      ypbind
e2image         initlog      lvmdiskscan  mount.ntfs-fuse  pvcreate       stop
[liveuser@localhost sbin]$ pwd
/sbin
[liveuser@localhost sbin]$ cd ../selinux
[liveuser@localhost selinux]$ ls
access    checkreqprot         compat_net  deny_unknown  initial_contexts  mls                   policyvers      user
avc       class                context     disable       load              null                  reject_unknown
booleans  commit_pending_bools create      enforce       member            policy_capabilities   relabel
[liveuser@localhost selinux]$
```

**Figure 7. A Screenshot View of a Deployed Release Configuration of the Alternative OA System Configuration Resulting From System Evolution**
(Scacchi & Alspaugh, 2012)

***Component License Evolution***—The license under which a component is available may change, as for example when the Mozilla core components changed from dual licensing to the tri-license (MPL, GPL, LGPL). Similarly, when Oracle Corporation took ownership of the Hudson CI system (Krill, 2011), the changes in intellectual property ownership and branding precipitated a major code fork and instigated parallel independent projects for sustaining development of this OSS CI system (Hudson-ci, 2011; Jenkins, 2013). Such evolutionary changes, which are common to OSS components, may require reconfiguring an OA system to migrate to a new (re-licensed) component version, or to an alternative system configuration (Scacchi & Alspaugh, 2012).

***In Response to Different Desired Rights or Acceptable Obligations***—The OA system's integrator or consumers may desire additional license rights (for example the right to sublicense in addition to the right to distribute) or no longer desire specific rights, or the set of license obligations they find acceptable may change. In either case, the OA system evolves in response, whether by changing components, evolving the architecture, or other means, to provide the desired rights within the scope of the acceptable obligations.

***Rapid Dynamic System Reconfiguration***—More advanced evolution scenarios entail support for building and releasing of multi-variant system deployment configurations that substitute functionally equivalent software component compilations that produce multiple, diverse executable binary images, each of which may execute in its own processor core, in a multi-threaded, multi-core processor (Jackson et al., 2011). Pursuing this new path requires a new compilation and build system regime, that in turn anticipates a new generation of CI and CD systems as future research subjects.

As should be clear, our purpose is not to provide a prescriptive model of the OA system evolution process, but instead to illuminate how different OA system evolution paths and activities point to issues in process specification, process integration, and the integration of different process enactment representations and mechanisms that must span/link manual-to-automated process hand-offs.

## Overall OA Development and Evolution Process Issues

Following from the software processes we examined in our case study and our review of related efforts, we see a number of issues for new software process research emerging. At least six such issues can be identified as follows.

First, we find that a central goal of process automation with widely available software integration and release deployment tools is to find enactment errors and articulation problems more quickly, rather than to provide prescriptive process guidance. Such process enactment details cannot be easily anticipated in general, so process specification and enactment must rely on trial and error, as well as process discovery (Jensen & Scacchi, 2006) to surface where additional or new process knowledge is to be found. Consequently, it is not surprising to observe the rise of a new class of software developer role, as “build-meisters”—developers who specialize in addressing the intricacies, quirks, and problems that arise during software integration processes, since such processes remain ad hoc, undefined, and difficult to model or improve.

Second, current continuous software development systems embody process specifications that are opaque, lack generality, and rely on the processing capabilities of specific incorporated tools to structure process enactment actions, decisions, and outcomes. Different CI systems embody different versions or variants of software build, test, and package processes. This implies that merely having a “defined” process model for processes like continuous integration and release deployment means that such a model will either be insufficiently detailed to provide anything beyond introductory level guidance, or more completely detailed but idiosyncratic because it is bound to specific process automation tools. This in turn makes the process specification problematic to adapt and evolve. There is a basic need for richer process models that represent both the idiosyncratic details of process automation tools for continuous integration and release management, and the generalized abstractions of such processes that can be reused for process (design) guidance and tailoring in specific software development organization settings (cf. Nichols et al., 2011).

Third, a recurring challenge from a process research standpoint is how to specify, model, analyze, or simulate software processes that span from mostly manual to mostly automated process enactment activities.

Fourth, automated process enactment systems are themselves subject to continuous improvement and evolution. This means the processes being supported are potentially evolving. However, if their process specification or model is tacit, or is encoded in implementation details, then the process may be opaque to all except the tool's developers. Thus, trying to specify, model, or simulate software processes that employ automated enactment systems requires the ability to address processes that are co-evolving, that is, how tool evolution drives development process evolution and how development process evolution precipitates tool evolution (cf. Scacchi, 2006). So choosing to attend only to one misses observation or specification of activities that enable or constrain the other. Such a dilemma points to another challenge for new software process research.

Fifth, process guidance specification and enactment automation are easily conflated in continuous integration and release deployment systems. As a result, developers of OA systems rely on informal best practices to get continuously-integrated software products out the door. Separating the specification of such processes from their implementation within the automated system would be an important contribution to the advancement of such systems. Similarly, providing guidance for how to specify processes more abstractly than as low-level process execution script commands (cf. Hypertable, 2013) would also contribute to the advancement of automated continuous software development systems.

Sixth, the development and evolution of component-based OA systems is both an interesting and a challenging problem for the software process research community. Such systems are likely to follow continuous software processes—processes that are repeatedly enacted hundreds to thousands of times during the sustained life of the system. Such processes are thus appropriate for careful empirical study, simulation, and analysis. The need to address how to continuously secure OA systems further complicates the challenges for software process research. Process streamlining optimizations, opportunities, and guidelines are likely subjects for further research and practical application. Similarly, when the software processes for securing an OA system involve automated process enactment, it appears that compliance testing—checking whether an automated enactment produced a system configuration that is compliant with the system's security policy—will increase in importance. Such compliance is likely to be ad hoc, unless the security policy is formalized into a computational model (Scacchi & Alspaugh, 2013) that can be cross-checked with the enactment results.

Last, empirical study of the software processes of interest, especially as they are observed in different OSS development projects, provides many insights and best practices that can help in the specification (modeling) and integration of processes for developing and evolving secure OA software systems.

## Conclusion

Process models provide a valuable means for specifying complex software production processes. Such models may have their greatest impact for project and process management, and for coordinating disparate software production processes together with automated enactment tools spread across an ecosystem of software producers. Explicit, open, and sharable process specifications are key to realizing these potential benefits, while the absence of such specifications means lost opportunities to reduce overall software production costs, improve software quality and security, and to streamline and continuously improve such explicit processes.

Managing and coordinating the development and evolution processes for producing secure open architecture software systems is challenging as we have shown in our case study. But as we have observed in our case study, widely available automated technologies for continuous integration and release deployment obscure or hide what these processes are. Further, we find that frequent errors and articulation problems in automated process enactment are expected, since process enactment details are ad hoc and idiosyncratic, while enactment processes are underspecified, not explicit, and encoded in an enactment system's implementation. However, automated process enactment systems may offer the potential to be extended to support (partially) automated process discovery and computational re-enactment (cf. Jensen and Scacchi, 2006), rather than just traditional process modeling and simulation. Thus, software producers of contemporary component-based OA systems are working against their self interests, assuming their interests are to improve their productivity and software quality, while reducing avoidable rework and other software production cost drivers.

Our study in this paper sought to identify a range of emerging issues in software process research, especially for process specification/modeling, as well as for process design, automation, and integration. Similarly, our case study highlights a number of ways how the need to continually secure an evolving OA system further complicates challenges for software process research. Finally, assuring that software development and evolution processes comply with extant system (or enterprise) security policies—which are presently informal requirements specification documents—means that process compliance checking arises as a practical need unmet by available software process tools.

Overall, our goal in this paper was to employ a case study and related research to help identify and articulate an emerging set of challenges for further software process research and development, Through both a review of related efforts and our case study, we identified a number of challenges for software process research whose investigation and resolution can lead to more streamlined and easier to continuously improve software development and evolution practices that are configured for specific organizations, different development tool chains, alternative target system platforms, and secure OA software product families, as well as for their evolutionary reconfiguration.

## Acknowledgments

This research is supported by grant #N00244-12-1-0067 from the Acquisition Research Program at the Naval Postgraduate School, and by grant #1256593 from the U.S. National Science Foundation. No review, approval, or endorsement is implied.

## References

Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2012). Software licenses, open source components, and open architectures. In I. Mistrík, A. Tang, et al. (Eds.), *Aligning enterprise, system, and software architectures* (pp. 58–79). IGI Global.

Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2013). The challenge of heterogeneously licensed systems in open architecture software ecosystems. In S. Jansen et al. (Eds.), *Software ecosystems: Analyzing and managing business networks in the software industry*. Edward Elgar Publishing.

Brown, A., & Wilson, G. (Eds.). (2012). *The architecture of open source applications*. Lulu.com.

Crnkovic, I., Chaudron, M., & Larsson, S. (2006). Component-based development process and component lifecycle. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA '06)* (pp. 44–54).

Defense Information Systems Agency [DISA]. (2012, October). *Network/perimeter/wireless—wireless (smartphone/tablet)* [Computer software manual]. Retrieved from http://iase.disa.mil/stigs/net_perimeter/wireless/smartphone.html

Dignan, L. (2012). Knight Capital future in jeopardy over botched software upgrade. *ZDNet*. Retrieved from http://www.zdnet.com/knight-capital-future-in-jeopardy-over-botched-software-upgrade-7000002116/

Duvall, P., Matyas, S., & Glover, A. (2007). *Continuous integration: Improving software quality and reducing risk*. Addison-Wesley Professional.

Fowler, M. (2000, September). *Continuous integration (original version)*. Retrieved from http://martinfowler.com/articles/originalContinuousIntegration.html

Garg, P. K., Mi, P., Pham, T., Scacchi, W., & Thunquest, G. (1994). The SMART approach for software process engineering. In *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)* (pp. 341–350).

Gizzi, N. (2011, May). Command and Control Rapid Prototyping Continuum (C2RPC) transition: Bridging the valley of death. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (pp. 135–154).

Gorlick, M. M., Strasser, K., & Taylor, R. N. (2012). Coast: An architectural style for decentralized on-demand tailored services. In *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA-ECSA '12)* (pp. 71–80).

Gray, J. (1981). The transaction concept: Virtues and limitations. In *Proceedings of the Seventh International Conference on Very Large Data Bases (VLDB '81)* (pp. 144–154).

Hudson-ci. (2011, August). *Hudson best practices*. Eclipsepedia. Retrieved January 2, 2013, from http://wiki.eclipse.org/Hudson -ci/Hudson_Best_Practices

Humble, J., & Farley, D. (2010). *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Addison-Wesley.

Hypertable. (2013). *How to build Hypertable on various platforms*. Retrieved December 1, 2012, from https://code.google.com/p/hypertable/wiki/HowToBuild

IBM Software Group. (2007). *SW5706 installation wizard hangs*. Retrieved from http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/topic/com.ibm.iea.was_v6/waspdguide/6.0/GettingStarted/Case2_Install _Wizard_Hangs.pdf

Jackson, T., Salamat, B., Homescu, A., Manivannan, K., Wagner, G., Gal, A., et al. (2011). Compiler-generated software diversity. In S. Jajodia, A. K. Ghosh, et al. (Eds.), *Moving target defense: Creating asymmetric uncertainty for cyber threats* (pp. 77–98). Springer. Retrieved from http://dx.doi.org/10.1007/978-1-4614-0977-9_4

Jenkins. (2013, January). *Upgrading from Hudson to Jenkins*. Retrieved from https://wiki.jenkins-ci.org/display/JENKINS/Upgrading+from+Hudson+to+Jenkins

Jensen, C., & Scacchi, W. (2005). Process modeling across the Web information infrastructure. *Software Process: Improvement and Practice, 10*(3), 255–272.

Jensen, C., & Scacchi, W. (2006). Experiences in discovering, modeling, and reenacting open source software development processes. In *Unifying the Software Process Spectrum* (pp. 449–462). Springer.

Krill, P. (2011, May). Oracle hands Hudson to Eclipse, but Jenkins fork seems permanent. *InfoWorld*. Retrieved from https://www.infoworld.com/d/application-development/ oracle-hands-hudson-eclipse-jenkins-fork-seems-permanent-021

Madhavji, N. H., Fernandez-Ramil, J., & Perry, D. E. (Eds.). (2006). *Software evolution and feedback: Theory and practice*. Wiley.

Mi, P., & Scacchi, W. (1991). Modeling articulation work in software engineering processes. In *Proceedings of the First International Conference on Software Process,* IEEE Computer Society (pp. 188–201).

Mi, P., & Scacchi, W. (1992, March). Process integration in CASE environments. *IEEE Software, 9*(2), 45–53.

Narayanaswamy, K., & Scacchi, W. (1987). Maintaining configurations of evolving software systems. *IEEE Trans. on Software Engineering, 13*(3), 324–334.

Nichols, W. R., Kirwan, P., & Andelfinger, U. (2011). A manifesto for effective process models. In *Proceedings of the 2011 International Conference on Software and Systems Process (ICSSP '11)* (pp. 242–244).

Qureshi, M. R. J., & Hussain, S. A. (2008). A reusable software component-based development process model. *Advances in Engineering Software, 39*(2), 88–94.

Scacchi, W. (2006). Understanding open source software evolution. In N. H. Madhavji, J. Fernandez-Ramil, & D. E. Perry (Eds.), *Software evolution and feedback: Theory and practice* (pp. 181–206). Wiley.

Scacchi, W., & Alspaugh, T. A. (2012, July). Understanding the role of licenses and evolution in open architecture software ecosystems. *Journal of Systems and Software, 85*(7), 1479–1494.

Scacchi, W., & Alspaugh, T. A. (2013). Advances in the acquisition of secure systems based on open architectures. *Cyber Security and Information Systems Journal, 1*(2). (To appear).

Scacchi, W., Brown, C., & Nies, K. (2012). Exploring the potential of virtual worlds for decentralized command and control. In *Proceedings of the 17th International Command and Control Research and Technology Symposium (ICCRTS)*.

Smalley, S. (2012, February). *The case for Security Enhanced (SE) Android.* 2012 Android Builder's Summit. Retrieved from https://events.linuxfoundation.org/images/stories/pdf/lf_abs12_smalley.pdf

Smith, P. (2011). *Software build systems: Principles and experience*. Addison-Wesley Professional.

*Thoughtworks CI feature matrix*. (2012). Retrieved January 2, 2013, from http://confluence.public.thoughtworks.org/display/CC/CI+Feature+Matrix

US-CERT. (2011). *Architecture and design considerations for secure software development*. U.S. Department of Homeland Security. Retrieved from https://buildsecurityin.us-cert.gov/swa/downloads/Architecture_and_Design_Pocket_Guide_v1.3.pdf

Continuous integration. (2012, December). In *Wikipedia*. Retrieved from http://en.wikipedia.org/wiki/Continuous_integration

*WineHQ*. (2013). Retrieved from http://winehq.org

*Xen hypervisor project*. (2013). Retrieved from http://xen.org/products/xenhyp.html

THIS PAGE INTENTIONALLY LEFT BLANK

# Challenges in the Development and Evolution of Secure Open Architecture Command and Control Systems

**18th International Command and Control Research and Technology Symposium**

**"*C2 in Underdeveloped, Degraded and Denied Operational Environments*"**

**Paper-ID-098**

**Topics**
Architectures, Technologies, and Tools (Primary)
Approaches and Organization (Secondary)

## Abstract

We identify challenges that arise during development and evolution of secure open architecture (OA) command and control (C2) systems. OA systems are those whose software system components and interconnection mechanisms are either proprietary closed source software offerings with open interfaces (e.g., application program interfaces), open source software, or some architectural configuration of closed and open source elements. Secure OA systems are those where the security of individual software elements may be uncertain, because of the ongoing evolution, poorly understood system integration compromises, or obtrusive software intellectual property licenses, yet where overall OA security must be continuously assured. We present a framework that organizes OA system security elements and mechanisms in forms aligned with stages of the life cycle of C2 for system design, building, and runtime deployment, as well as system evolution. We provide a case study to show our scheme and how it can be applied to C2 system architectures that rely on an OA. Finally, we show how our efforts complement and extend the agile C2 framework that utilizes a new generation of software components and security mechanisms that are engineered and adapted by multiple parties and disseminated within a diverse marketplace ecosystem of software producers, integrators, and consumers.

## Introduction

In this paper, we identify and investigate technical and acquisition challenges that arise during the development and evolution of secure open architecture (OA) command and control (C2) systems. OA systems are those whose software system components and interconnection mechanisms are either proprietary closed source software offerings with open interfaces (e.g., application program interfaces), open source software (including government open source software or defense open source software (OSS) via Forge.mil) or some architectural configuration of closed and open source elements. Secure OA systems are those where the security of individual software elements may be uncertain, because of the ongoing evolution, poorly understood system integration compromises, or obtrusive software intellectual property licenses, yet where overall OA security must be continuously assured (Scacchi & Alspaugh, 2012a, 2012b, 2012c).

It is now clear that future C2 systems must resist internal or external attacks on single and multiple system components, interconnection interfaces, or data repositories. No longer can we rely on air-gap system barriers, or security through proprietary obscurity, as individual security barriers can be compromised through intrusive cyberwarfare software attack vectors or social engineering. Furthermore, current approaches to system security are most often piecemeal with little or no support for guiding what system security requirements must address across different software system processing elements and data levels, and how those can be manifest during the design, building, deployment, and evolution of OA software systems. Finally, agile C2 efforts seek to transform overall system development and evolutionary adaptation time frames from years to months (or less; Reed, Benito, Collens, & Stein, 2012). This means fundamentally new approaches to secure C2 system development and evolution must be available.

We present a framework that organizes OA system security elements and mechanisms in forms that can be aligned with different stages of the life cycle of C2 system design, building, and run time deployment, as well as system evolution. We provide a case study to show our scheme and how it can be applied to centralized C2 system architectures like C2RPC (Garcia, 2011; Gizzi, 2011) or to next-generation decentralized C2 systems (Scacchi, Brown, & Nies, 2012) that rely on an OA. Finally, we show how our efforts complement and extend the agile C2 framework that utilizes a new generation of software components (apps, widgets, connectors, and encapsulation mechanisms) that are sourced from a diverse marketplace ecosystem of software producers (Reed et al., 2012).

## Open Architectures for Command and Control Systems

Open architecture (OA) software is a customization technique that enables third parties to modify a software system through its exposed architecture, evolving the system by replacing its components, connectors, or configuration. The three military services within the U.S. Department of Defense are pursuing initiatives that encourage the adoption of OA approaches and OA systems as way to reduce system development costs over the life of a system (Acquisition Community Connection [ACC], 2013). Increasingly more software-intensive systems are developed using an OA strategy, not only with proprietary closed source software components with open APIs, but also with OSS components. However, composing a system with components that are subject to different intellectual property (IP) licenses increases the likelihood of conflicts, liabilities, and no-rights stemming from incompatible licenses. We call systems whose components are subject to different licenses *heterogeneously-licensed systems* (Alspaugh, Scacchi, & Asuncion, 2010). So in our work we define an OA C2 system as a software system for command and control consisting of components that are either OSS or proprietary with open APIs, whose overall system license rights at a minimum allow its use and redistribution, in full or in part. It may appear that using a system architecture that incorporates OSS components and uses open APIs will result in an OA system. But not all such architectures will produce an OA, since the (possibly empty) set of available license rights for an OA system depends on (a) how and why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, and (c) the degree to which the licenses of different OSS components encumber all or part of a software system's architecture into which they are integrated (Alspaugh et al., 2010; Scacchi & Alspaugh, 2012; AAS12).

The following kinds of software elements appearing in common software architectures can affect whether the resulting systems are open or closed (Bass, Clements, & Kazman, 2003).

***Software Source Code Components***—These can be either (a) standalone programs, (b) libraries, frameworks, or middleware, (c) inter-application script code such as C shell scripts, or (d) intra-application script code, as for creating rich Internet applications using domain-specific languages such as XUL for the Firefox Web browser, "mashups," or their composition into widgets (Feldt, 2007; Soylu et al., 2011; OWI3). Their source code is available, and they can be rebuilt. Each may have its own distinct license.

***Executable Components***—These components are in binary form, and the source code may not be open for access, review, modification, or possible redistribution. If proprietary, they often cannot be redistributed, and so such components will be present in the design- and run-time architectures but not in the distribution-time architecture.

***Software Services***—An appropriate network-accessible software service can replace a source code or executable component.

***Application Programming Interfaces/APIs***—Availability of externally visible and accessible APIs is the minimum requirement for an "open system" (Meyers & Oberndorf, 2001). Open APIs are not and cannot be licensed, and can limit the propagation of license obligations.

***Software Connectors***—Software whose intended purpose is to provide a standard or reusable way of communication through common interfaces, e.g., Microsoft.NET, Enterprise Java Beans, GNU Lesser General Public License (LGPL) libraries, and data communication protocols like the Hypertext Transfer Protocol (HTTP). Connectors generally limit the propagation of license obligations.

***Methods of Connection***—These include linking as part of a configured subsystem, dynamic linking, client–server connections, and what we call "interface shims" (abstract interfaces or interface libraries). Methods of connection affect license obligation propagation, with different methods affecting different licenses.

***Configured System or Subsystem Architectures***—These are software systems that are used as atomic components of a larger system, or as a reusable or "functional capability," such that its internal architecture may be comprised of components with different licenses, affecting the overall system license. To minimize license interaction, a configured system or sub-architecture may be surrounded by what we term a license firewall, namely a layer of dynamic links, client–server connections, license interface shims, or other connectors that block the propagation of reciprocal obligation. Similarly, a configured system or subsystem can be encapsulated within a security mechanism such as a virtual machine (Xen.org, 2012).

Examples of such elements appear in descriptions and figures presented later in this paper. But the diversity of the kinds of elements that appear in an OA system enables the design, development, and evolution of agile C2 systems within an agile and adaptive software ecosystem (Reed et al., 2012), as we well show.

## Accommodating Agile C2 Development and Evolution

The MITRE Corporation and others in the defense community seek to embrace the development of agile C2 systems (Reed et al., 2012). Such systems are envisioned to arise from the assembly and integration of system elements (such as application components, widgets, content servers, networking elements) within a software ecosystem of multiple producers, integrators, and consumers who may supply or share the results of their efforts. The assembly and integration of system elements produces "C2 system capabilities" (C2SCs). C2SCs may be produced, acquired, integrated, shared, or reused by different trusted parties. C2SCs may address a set of ISR data/signal processing components, office productivity components supporting mission planning, or the like. Our purpose is to identify how our approach to the design of secure OA systems can be aligned with their vision for agile C2 systems. Along the way we focus on design of OA system capability involving office productivity components that must be configured as a secure C2SC.

The design and development of agile C2 systems follows from two sets of principals: one set addressing guidelines and tenets for multi-party engineering (MPE) of C2 system components, and the other set addressing attributes of agile and adaptive ecosystems (AAE) for producing C2SCs or C2 system elements. For brevity, we simply identify these principles for MPE and AAE, as they are more fully explained elsewhere (Reed et al., 2012), but we do so in ways that foreshadow and more clearly align with our approach that follows in a later section.

***MPE Tenets:***

1. Provide small system components that can be rapidly developed and accommodate different functionally equivalent variants or functionally similar versions
2. Certify components are consistent with "shared agreements" regarding security requirements, system architecture, data semantics, production and integration processes or process constraints, and other aspects of mission-specific or mission-common domain models
3. Supply diverse C2 system components via a market of component producers or integrators
4. Assemble and integrate C2SCs from components available in the market that are consistent with relevant shared agreements
5. Provide feedback from C2 system users to component producers or capability integrators to improve market efficiency and effectiveness

***AAE Attributes:***

1. Encourage and sustain a software ecosystem that is agile (supports assembly and integration C2SC) from components in market, and adaptive (supports substitution of functionally similar component versions or functionally equivalent component variants), in line with user feedback.
2. Component markets are federated so as to accommodate sharing, reuse, or trading of components across different system integrators or consumer organizations.
3. Shared agreements serve as a basis for enabling multi-party collaboration in system development, integration, and evolution/sustainability.
4. Production, integration, or post-deployment support for components or C2SCs must be viable for small businesses or large, as well as promoting market diversity and effectiveness.

5. Consumer/user organizations seek to manage portfolios of components or C2SCs that collectively improve mission effectiveness, agility, and adaptiveness, while reducing costs.

Finally, to help understand what we mean by a software ecosystem, we use Figure 1 to represent where different parties are located across a generic software ecosystem and the supply networks or multi-party relationships that emerge to enable the software producers to develop and release products that are assembled and integrated by system integrators for delivery to consumer and end-user organizations.

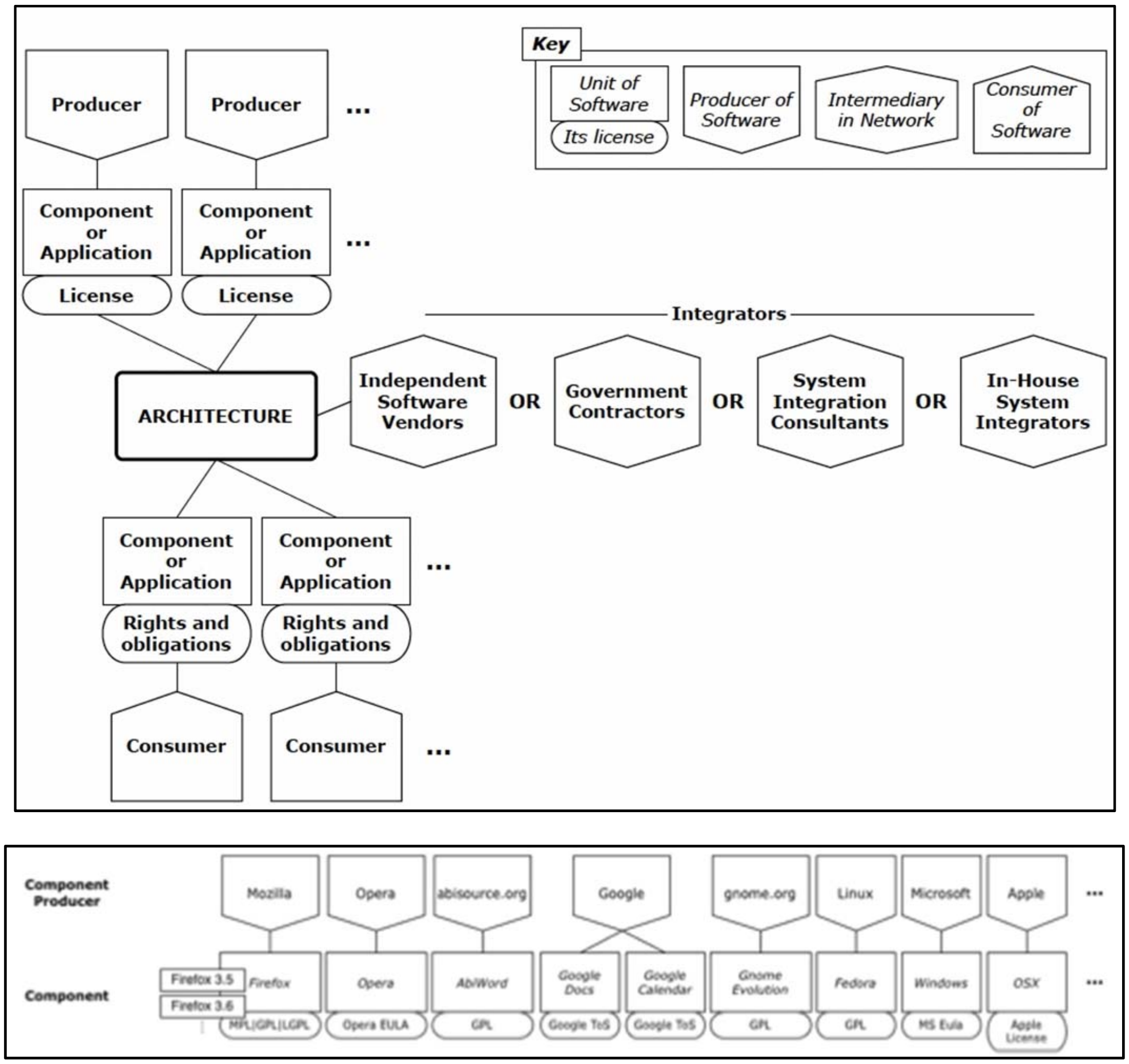


**Figure 1. A Generic Software Ecosystem Supply Network (Upper Part), Along with a Sample Elaboration of Producers, Software Components, and Licenses for an OA System Components They Employ (Lower Part)**
(Scacchi & Alspaugh, 2012)

The lower part of Figure 1 also identifies where elements of shared agreements like IP licenses enter into the ecosystem and how the assembly of components into a configured system or subsystem architecture by system integrators effectively (and perhaps unintentionally) determines which IP license obligations and rights get propagated to consumer and end-user organizations. Agreement terms and conditions acceptable to consumer/end-user organizations flow back to the integrators. This helps reveal where and how shared agreements will mix, match, mashup, or not at the system architecture level, which is another reason for why we use (and advocate) explicit OA system models.

## A Framework for Securing Agile OA C2 Systems

Over the past five or so years, we have been researching, prototyping, and refining an approach to the acquisition, development, and evolution of OA software systems (Alspaugh et al., 2010; Scacchi & Alspaugh, 2012; AAS12). Central to our approach is our reliance on explicit models of software system architectures described in an architectural description language (Taylor, Medvidovich, & Dashofy, 2009). Use of explicit architectural models is key to making OA systems tractable and observable, since we want to be able to make visible where and how a system's architecture is open and where it is not, during different system development and evolution activities. Explicit architectural representations are also key to coordinating the development, integration, deployment, and evolution of complex OA software systems among a dispersed community of development and user organizations (Ovaska, Rossi, & Marttiin, 2003). Our models also draw attention to the identification of a system's elements and their configuration into a system capability or complete system. In addition, our models allow for the system elements to be specified by type or instance (e.g., Web browser, Microsoft Internet Explorer), as well as optionally specifying functionally similar versions or functionally equivalent variants thereof (e.g., Internet Explorer 8 and Internet Explorer 9 are similar, while 32-bit and 64-bit variants of IE9 are equivalent).[1] Finally, we annotate our models with formal expressions that allow us to specify details like IP license or security policy obligations and user rights in ways that are amenable for acquisition contracting and auditing, and compliance practices (Alspaugh et al., 2010). Thus, our annotated OA system models form a core of the shared agreements identified as a key element to the development of agile C2 systems within an agile and adaptive ecosystem (Reed et al., 2012). The remainder of this section identifies other aspects of our approach that align with the MPE/AAE framework.

Let us consider what needs to be specified during the acquisition of an OA C2 system that allows for a system suggested by use within the command and control rapid prototyping continuum, C2RPC (Garcia, 2011; Gizzi, 2011). Such a system incorporates both mission-specific components (applications or widgets for processing ISR data, e.g., Gerschefske & Witmer [2012]) and also common office productivity applications that run on a personal computer networked to remote servers. Such a system can include a Web browser, word processor, email and calendaring applications that are configured to operate on a personal computer, where the PC's operating system, Web browser, and other applications need to be

[1]Many software producers utilize multi-level numerical identifiers or other nomenclature (e.g., Internet Explorer 10 Release Preview) to distinguish major version releases from minor revision variants (e.g., Internet Explorer 9 versus Internet Explorer 9.1.3) which we also accommodate. Such specificity is required to support system integration and deployment requirements.

configured to access remote data/Web content servers. Figure 1 shows part of the system ecosystem of software producers and the components they can provide for our enterprise system. A Web browser like Mozilla Firefox, Microsoft Internet Explorer, Opera, or Google Chrome can further be tailored and invoked through internal scripts to support small, mission-specific widgets, as might be developed using the Ozone Widget Framework (Ozone Widget Framework, 2013).

Figure 2 shows the reference design of an OA system architecture of the office productivity capability associated with a C2 system (cf. Garcia [2011]). This OA system design also accommodates the integration of browser-based remote networked services or scripted widgets. What might a secure software product line for a system like this involve, and how might it provide benefits and security qualities to be specified for design time, build time, and run time? How can its OA and product-line characteristics contribute to security throughout the acquisition system life cycle?

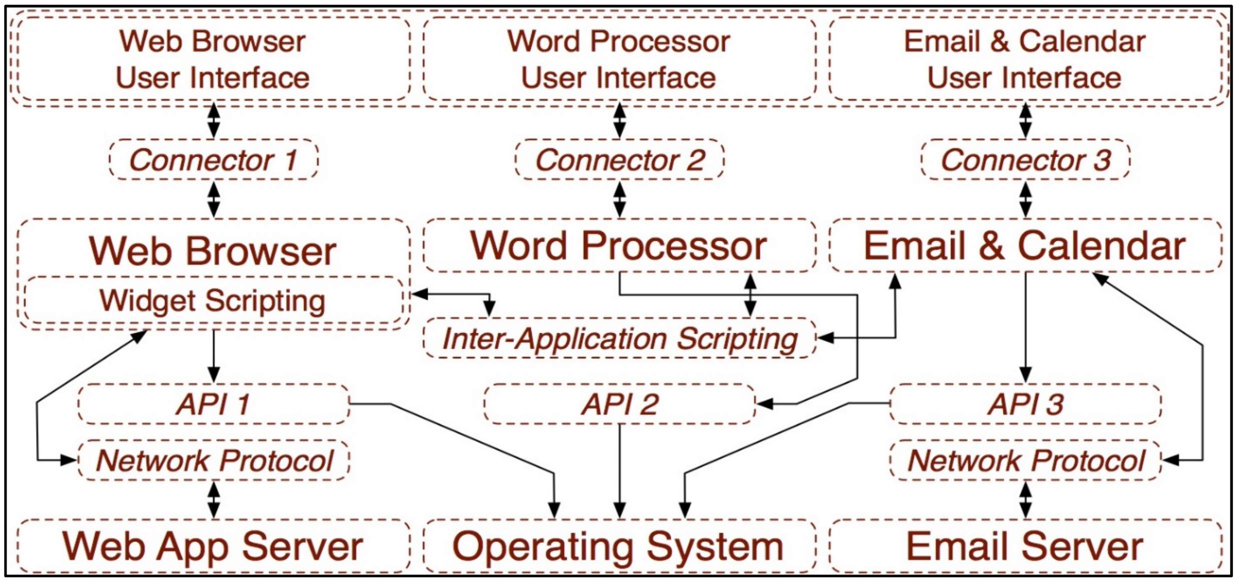


**Figure 2. A Design-Time Reference Model of an OA System That Accommodates Multiple Alternative Software Component Selections and Configurations**

Within our approach, we address non-functional C2 system requirements, such as security, configurability into C2SC, and post-deployment adaptation. These requirements are elaborated at design and integration times by specific functional requirements that explain how and to what degree the non-functional requirements are going to be satisfied at deployment time for consumer/end-user organizations (Alspaugh et al., 2010; Scacchi & Alspaugh, 2012; AAS09).

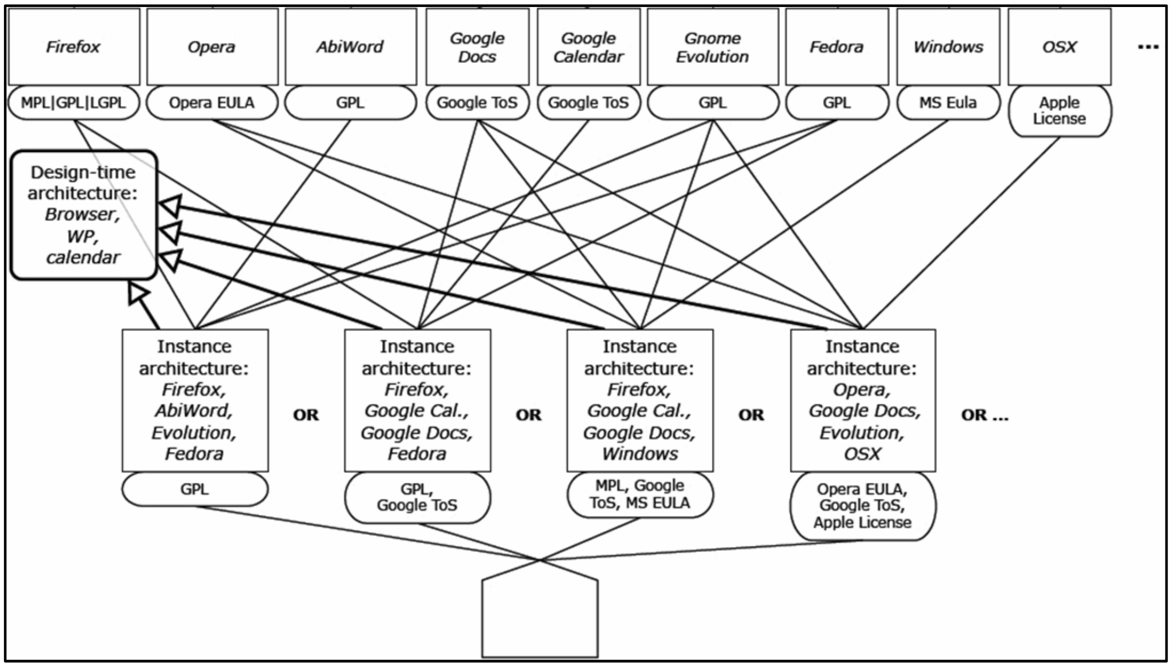

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
...
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
OR ...
GPL
GPL, Google ToS
MPL, Google ToS, MS EULA
Opera EULA, Google ToS, Apple License

**Figure 3. A View of an OA Software Ecosystem That Provides Alternative, Functionally Similar Components Compatible With the Reference Design-Time Architecture**

Figure 3 illustrates a possible software product line that follows from the OA software ecosystem shown in Figure 2. Here a number of possible producers and the components they produce and license come into play, within four specific instance C2SC architectures. With appropriate architectural topologies, and appropriate shim components and connectors inserted between the major components, each of these four instances supports the same general functionality of the office productivity C2SC that can support mission planning. This means that it becomes possible to offer support for rapidly switching from one OA system configuration to another by substituting compatible (functionally similar) components. This gives us the ability to adapt the C2SC in ways that sustain its overall operational requirements, while allowing multiple parties to independently maintain or evolve the component configuration they choose.

Last, it is also possible to achieve different nonfunctional requirements addressing support for security policies through the four architectural choices, for example, by requiring that computer operating systems on which such a capability is hosted must support an appropriate mandatory access control and type enforcement mechanism, such as is provided in the Security-Enhanced Linux protection service library (e.g., for computers running the RedHat Enterprise Linux or Fedora operating systems), or by requiring the use of secure network protocol connectors like HTTPS, which provide basic network data encryption functionality.

## A Case Study for Securing OA C2 Systems

We utilize a case study to describe and analyze our approach to design secure OA C2 systems or C2SCs. Such a study serves to help identify issues that arise pertaining to our support of MPE/AAE elements, which in turn drive the development or evolution of such systems, whether they are deployed in a fully developed environment, or whether they are envisioned to address challenges that arise in underdeveloped, degraded or resource-limited operational environments. Our study is divided into two parts, the first addressing design of a simple centralized C2 system with an OA for use in a fully developed environment, and the second addressing a similar but decentralized C2 system with an OA, which may be appropriate for experimental studies.

### Centralized C2 System Architecture

Traditional C2 systems are designed to support a centralized system deployment. In such a situation, all core system elements or capabilities are located in a single facility, though such a facility may be mobile (e.g., airborne or shipboard).

Within the overall ecosystem of Figure 3, Figure 4 shows one possible instance ecosystem involving specific producers (Mozilla—Firefox, abisource.org—AbiWord, gnome.org—Evolution, Red Hat—Fedora) and specific component alternatives selected (i.e., Firefox, AbiWord, Evolution, Fedora).

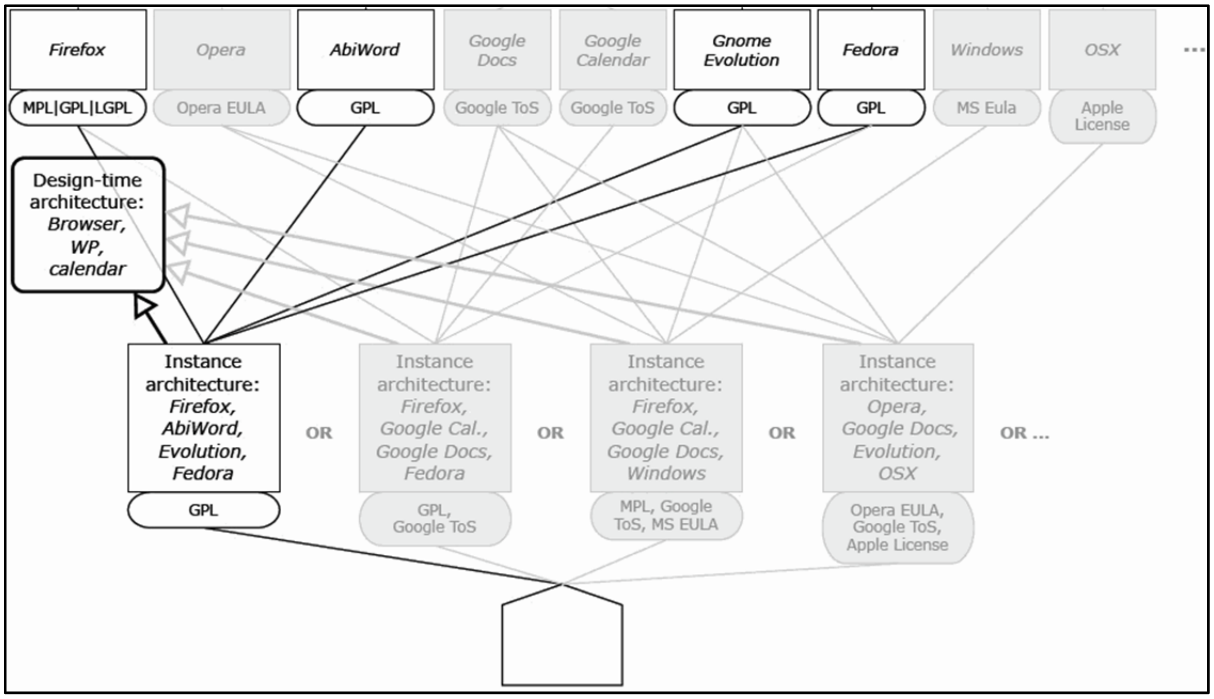
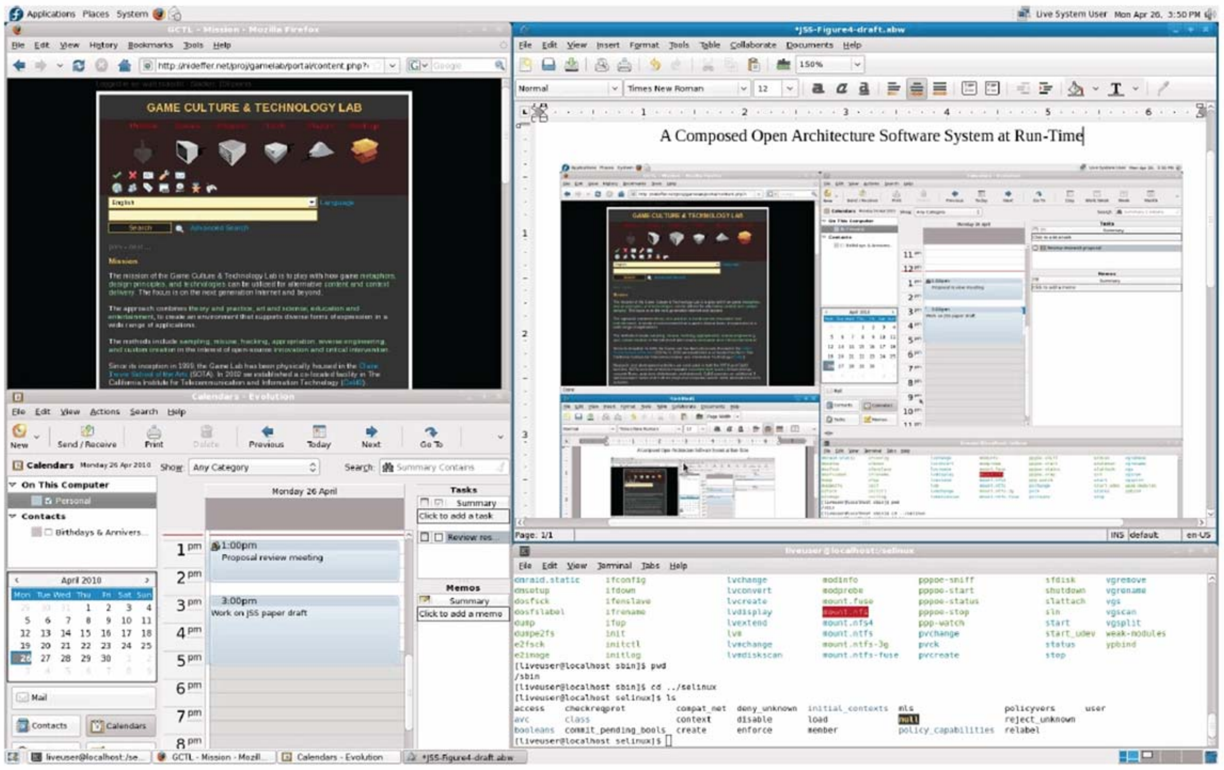

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
...
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
GPL
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
GPL, Google ToS
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
MPL, Google ToS, MS EULA
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
Opera EULA, Google ToS, Apple License
OR ...

**Figure 4. A Selection Among Alternative Components That Can Be Included at Build-Time to Produce an Integrated System Compatible With the Design-Time Reference**

Figure 5 then shows what a deployed run-time instantiation of this OA C2SC might look like from the perspective of the system's end-users. Here we see the Firefox Web browser (upper left corner), AbiWord word processor (upper right corner), Gnome Evolution email + calendaring application (lower left corner), and the Fedora operating system (lower right corner). Though the visual detail in this example is limited, the Red Hat Fedora Linux operating system (lower right corner) is shown utilizing the *SELinux* security protection library, for coding and enforcing mandatory access control on programs/data and other security enforcement functions.


GAME CULTURE & TECHNOLOGY LAB

Mission

The mission of the Game Culture & Technology Lab is to play with how game metaphors, design principles, and technologies can be utilized for alternative content and context delivery. The focus is on the next generation Internet and beyond.

The approach combines theory and practice, art and science, education and entertainment, to create an environment that supports diverse forms of expression in a wide range of applications.

The methods include sampling, misuse, hacking, appropriation, reverse engineering, and custom creation in the interest of open-source innovation and critical intervention.

Since its inception in 1999, the Game Lab has been physically housed in the Claire Trevor School of the Arts (SOTA). In 2002 we established a co-located facility in The California Institute for Telecommunication and Information Technology (Calit2).

A Composed Open Architecture Software System at Run-Time

```
dmraid.static    ifconfig      lvchange      modinfo         pppoe-sniff      sfdisk       vgremove
dmsetup          ifdown        lvconvert     modprobe        pppoe-start      shutdown     vgrename
dosfsck          ifenslave     lvcreate      mount.fuse      pppoe-status     slattach     vgs
dosfslabel       ifrename      lvdisplay     mount.nfs       pppoe-stop       sln          vgscan
dump             ifup          lvextend      mount.nfs4      ppp-watch        start        vgsplit
dumpe2fs         init          lvm           mount.ntfs      pvchange         start_udev   weak-modules
e2fsck           initctl       lvmchange     mount.ntfs-3g   pvck             status       ypbind
e2image          initlog       lvmdiskscan   mount.ntfs-fuse pvcreate         stop
[liveuser@localhost sbin]$ pwd
/sbin
[liveuser@localhost sbin]$ cd ../selinux
[liveuser@localhost selinux]$ ls
access    checkreqprot          compat_net  deny_unknown  initial_contexts  mls                   policyvers      user
avc       class                 context     disable       load              null                  reject_unknown
booleans  commit_pending_bools  create      enforce       member            policy_capabilities   relabel
[liveuser@localhost selinux]$
```

**Figure 5. An End-User Run-Time Version of the Selected Alternative Components That Fulfills the OA C2SC System Design**

Figure 6 outlines an alternative system configuration and the instance ecosystem that produces it. This instance architecture substitutes services for components in the case of Google Docs for the word processing functionality and Google Calendar for the calendar functionality. With appropriate shims and changes to the architectural topology, this combination of major components could also support the system's functional requirements, and because the services are accessed through client–server connections, which block the propagation of most license obligations, there are a number of ways to satisfy the IP constraints imposed by the component and service licenses.

This alternative configuration also highlights possible acquisition-time concerns and the nonfunctional requirements and security license issues that follow from them. For example, a remote service such as Google Docs provides benefits and imposes costs with respect to a compiled component such as AbiWord. On the one hand, the remote service makes some qualities easier to achieve (data sharing, backup, etc.) but on the other may make some qualities harder to achieve (data security over a network connection and in the "cloud," up-time of the service, little or no control over when new versions of the service are used compared to complete control over when new versions of a component are integrated).

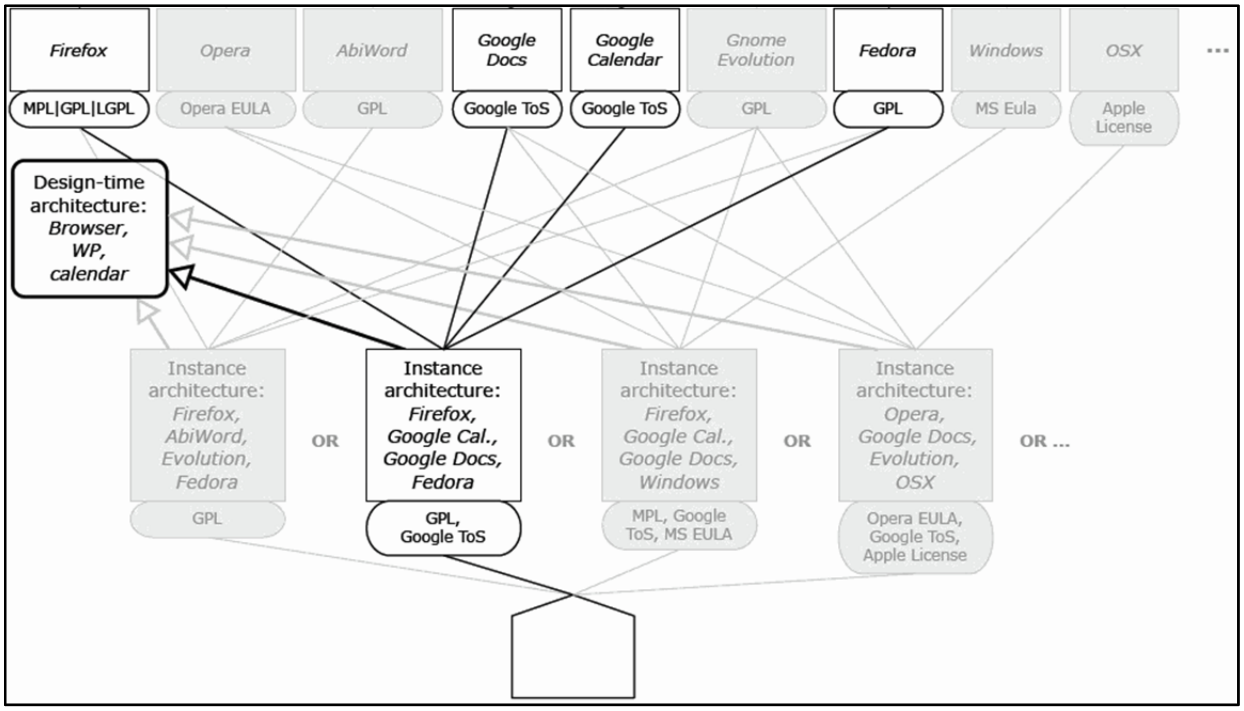
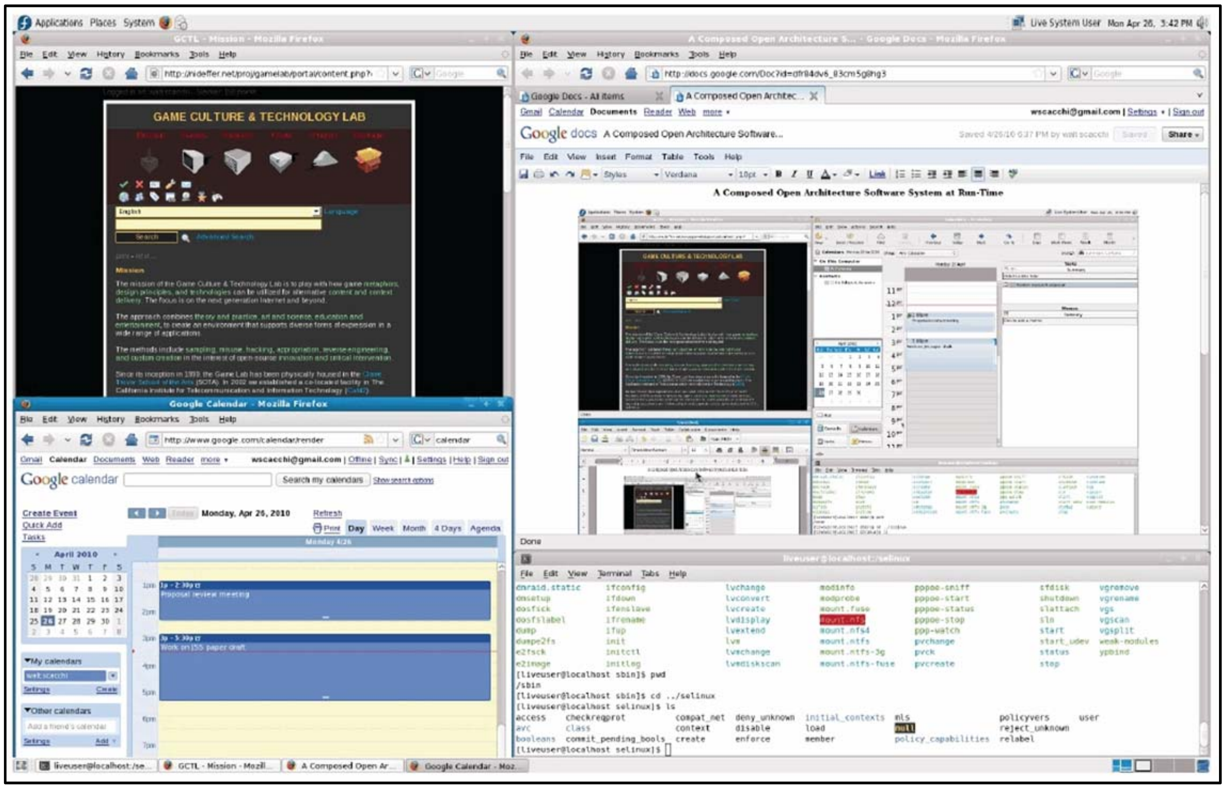

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
...
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
GPL
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
GPL, Google ToS
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
MPL, Google ToS, MS EULA
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
Opera EULA, Google ToS, Apple License
OR ...

**Figure 6. A Second Configuration of the OA C2SC Example, Using Alternative but Functionally Similar Components**

This ability to rapidly and conveniently substitute components (agility) to adapt to different end-user needs or required components aligns with the MPE/AAE elements. However, we still need to address how other elements in the shared agreements, like IP licenses, enable or constrain whether such an alternative configuration, though technically possible, meets other organizational requirements. This concern raises the following kinds of questions:

- Who in the ecosystem of human actors (the multiple parties) for this system has the right to make the decisions to use a remote/networked computational service in place of a locally hosted software system component, or one component version in place of another? What obligations are they required to satisfy first? These questions are of concern at acquisition time and, we claim, are addressable through explicit acquisition policies that stipulate desired rights and acceptable obligations by the acquiring organization, where such policies are important to system acquisition officers, just as IP licenses do for IP rights and obligations important to software producers. Our shared agreements need to provide guidance for what to do here.
- When can these decisions be made? In traditional development processes these would occur at design time, but in the larger view we examine here such decisions, or rather the shared agreements that

control them, are perhaps more properly considered at component, C2SC, or system acquisition time. As we will see below, it is also possible that in order to achieve specific security qualities they might be made at system integration time or at deployment time, in response to specific end-user organization needs.



```
dmraid.static   ifconfig     lvchange     modinfo          pppoe-sniff    sfdisk      vgremove
dmsetup         ifdown       lvconvert    modprobe         pppoe-start    shutdown    vgrename
dosfsck         ifenslave    lvcreate     mount.fuse       pppoe-status   slattach    vgs
dosfslabel      ifrename     lvdisplay    mount.nfs        pppoe-stop     sln         vgscan
dump            ifup         lvextend     mount.nfs4       ppp-watch      start       vgsplit
dumpe2fs        init         lvm          mount.ntfs       pvchange       start_udev  weak-modules
e2fsck          initctl      lvmchange    mount.ntfs-3g    pvck           status      ypbind
e2image         initlog      lvmdiskscan  mount.ntfs-fuse  pvcreate       stop
[liveuser@localhost sbin]$ pwd
/sbin
[liveuser@localhost sbin]$ cd ../selinux
[liveuser@localhost selinux]$ ls
access    checkreqprot         compat_net  deny_unknown  initial_contexts  mls                   policyvers      user
avc       class                context     disable       load              null                  reject_unknown
booleans  commit_pending_bools create      enforce       member            policy_capabilities   relabel
[liveuser@localhost selinux]$
```

**Figure 7. An End-User View of the Alternative Run-Time System Configuration**

Figure 7 shows a run-time view of this alternative configuration. To the end user this system appears quite similar to the one in Figure 5, and the differences might scarcely be noticed, which raises the next set of possibilities.

Both these instance architectures displayed in Figure 5 and Figure 7 specify specific alternatives for the major components, for example, Mozilla Firefox for the Web browser component. But which version of Firefox? For example, it is quite possible that both of the instance architectures discussed above could be implemented using either Firefox 18 or Firefox 19, satisfying all the functional requirements with no change to the instance architecture and no revision of software shims. Who has the power to decide to use version 18 rather than version 19? How late in the software process can this decision be made—for example, could it be made as late as system startup time by a system user or in response to a particular security attack on the previous configuration?

Finally, an orthogonal consideration is the use of software-based access control containment vessels to encapsulate components or subsystems within a virtual machine, to monitor and control interactions among components and subsystems in order to block attacks and protect vulnerable parts of a system. Figure 8 shows a screenshot in ArchStudio of a design-time architecture utilizing eight containment vessels, seven for individual components and connectors and the eighth for the group of components and connectors associated with the computer's operating system.

For security, the Fedora operating system can employ the SELinux capabilities to restrict all shell/operating systems commands through mandatory access control and type enforcement, while other components can all be contained within one (for minimal security confinement) or more (for increased security confinement on a per component basis) Xen-based virtual machines (See Figure 8). The interoperability of SELinux and Xen is now a common feature of many large Linux system installations (e.g., Amazon.com now has more than 500,000 Linux systems running Xen; Xen.org, 2012). So it is possible for shared agreements to call for the use of multiple software-based security mechanisms to protect a OA system or C2SC, while still accommodating the MPE/AAE elements. This is an important accomplishment.

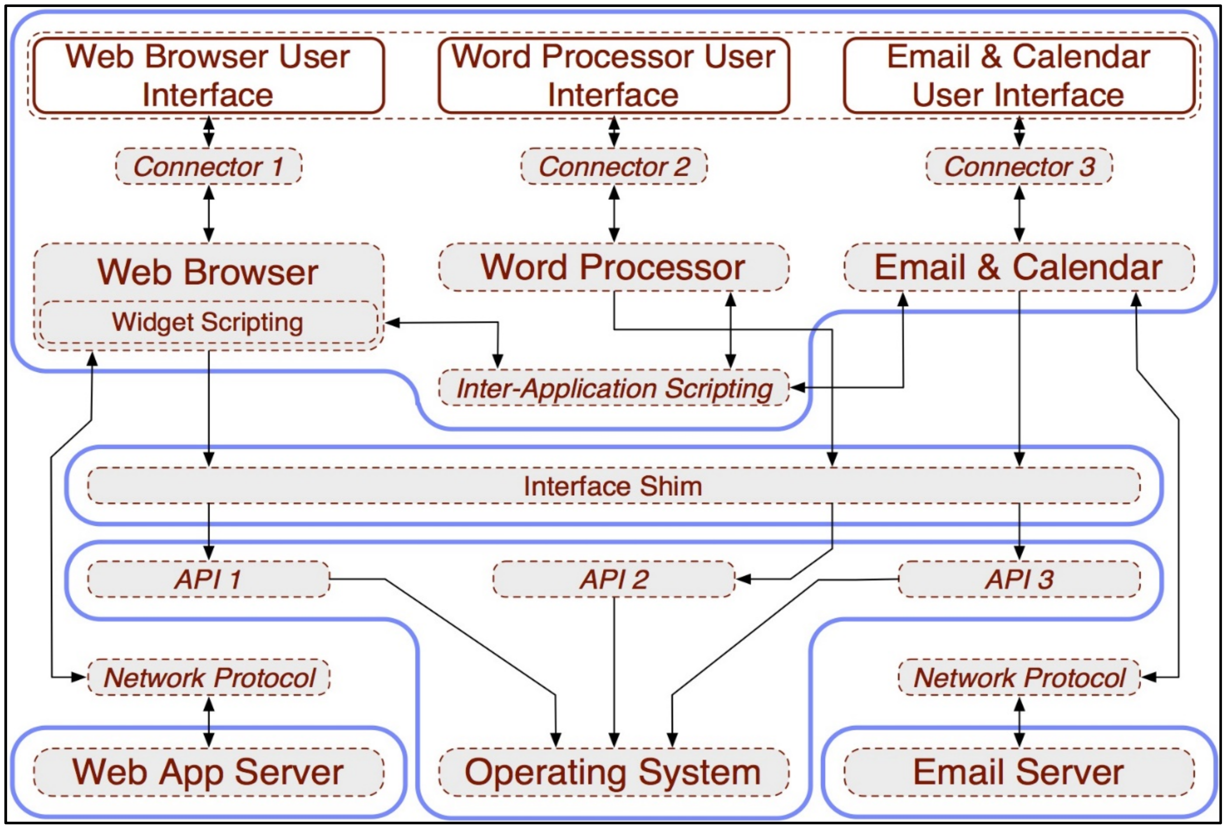


**Figure 8. A Secure OA System Configuration Alternative That Encapsulates Multiple System Capabilities Within Different Virtual Machines (e.g., Using [Xen.org, 2012]), Where Each System Capability May be Under the Purview of a Different Organizational Authority**

## Decentralized C2 System Architecture

Decentralized C2 systems can employ OAs that accommodate their deployment and usage in degraded and limited operation environments. Decentralized C2 systems can have a very small physical footprint, and mission planners/commanders may be located potentially anywhere in the world. So decentralized OA C2 systems can serve as appropriate candidates for experimentation or training in underdeveloped, degraded, or denied operational environments.

In our previous work, we have investigated and prototyped a C2 system called DECENT (Scacchi et al., 2012) that provides an immersive 3D virtual world for experimenting with decentralized C2 activities. DECENT is designed to run in a wide-area network environment that supports 3D browsers (or conventional Web browsers with 3D world viewer plug-ins) as clients, and remote servers to provide content and other services accessible through the browsers. DECENT operates on the Internet, but not on the Web, relying instead on a separate decentralized network grid architecture called the *HyperGrid* (Lopes, 2011). The HyperGrid infrastructure in turn allows for the establishment and operation of distinct or interconnected hypergrids as separate administrative authorities for research, experimentation, or training applications. The MOSES hypergrid is such an example (Maxwell, 2012; Military Open Simulator Enterprise Strategy [MOSES], 2013).

The Ozone Widget Framework (Ozone Widget Framework, 2013) supports a different form of decentralized OA. OWF provides support for the development of Web-compatible widgets as lightweight applications that access pre-specified kinds of content from remote servers. This allows the creation of virtual private networks offering Web-like applications and services through network-managed security capabilities. However, in an underdeveloped operational environment, such networking capabilities may not be available or reliable, so other means must be utilized to realize secure communications. Alternatively, applications or widgets that rely on signals from known types of sensors may also not be available or reliable, so other widget versions may be need to be provided.

Subsequently, one strategy for developing a decentralized C2 system would combine the OA for a centralized C2 system, but allow for the system components (applications, widgets, content servers, operating systems, etc.) or C2SCs to be interconnected to local/remote servers through encrypted network connectors (e.g., secure data communication protocols like HTTPS or TLS, or robust dynamic capability connectors, like those used in the COAST [Gorlick, Strasser, & Taylor, 2012]) that enable data, control, or signals to flow across (or tunnel beneath) virtual software defined networks. In other words, the user interface would still rely on a Web browser modality, yet in a simple implementation, be able to securely access hypergrid worlds in one window, with other widget-based content services located in other browser window or user sessions, depending on overall security policy. So we have a basis for developing a secure decentralized C2 system that allows a consistent OA system model whose components or connectors may be centralized or decentralized by design choice or security policy. However, compiling and deploying such a decentralized C2 system are the system development activities when the security components will be integrated.

Overall, a decentralized C2 system can be developed using system elements (components, connectors, or embedded C2SCs) that may reside on local computer or remote networked computers that are accessed through software client applications that may reside within a C2 virtual world. Such choices may be most appropriate for OA C2 systems that are intended to support experimentation in C2 mission planning activities. Such activities may be most relevant where underdeveloped or degraded system elements are employed, in order to help train mission commanders to learn how to articulate their requirements for new system components that can be rapidly developed, integrated, and deployed. It also can serve to reveal other practical advantages and constraints that arise when end-user organizations follow the MPE/AAE guidelines for adapting and evolving their OA C2 systems.

## Conclusions

In this paper, we identified and investigated technical and acquisition challenges that arise during the development and evolution of secure open architecture (OA) command and control (C2) systems. OA systems are identified as those whose software system components and interconnection mechanisms are either proprietary closed source software offerings with open application program interfaces, open source software, or some architectural configuration of closed and open source elements. Secure OA systems are identified as those where the security of individual software elements may be uncertain, because of the ongoing evolution, poorly understood system integration compromises, or obtrusive software intellectual property licenses, yet where overall OA security must be continuously assured.

We presented a framework that organizes OA system security elements and mechanisms in forms that can be aligned with different stages of the life cycle of C2 system development. We focused attention to the design of OA C2 systems or C2 system capabilities using commonly available software components that provide office productivity capabilities that support C2 operations. We provided a case study to show our scheme and how it can be applied to C2 system architectures that rely on an OA. Finally, we showed how our efforts complement and extend the agile C2 framework that utilizes a new generation of software components and security mechanisms that are engineered or adapted by multiple parties and disseminated within a diverse marketplace ecosystem of software producers, integrators, and consumers.

## Acknowledgements

The research described in this report was supported by grant #N00244-12-1-0004 from the Acquisition Research Program at the Naval Postgraduate School, Monterey, CA, and by grant #1256593 from the National Science Foundation. No endorsement, review, or approval implied. This paper reflects the views and opinions of the authors, and not necessarily the views or positions of any other group, enterprise, or government agency.

## References

Acquisition Community Connection (ACC). (2013). *Naval open architecture and DoD open systems architecture*. Retrieved from https://acc.dau.mil/oa

Alspaugh, T. A., Scacchi, W., & Asuncion, H. (2010). Software licenses in context: The challenge of heterogeneously-licensed systems. *Journal of the Association for Information Systems, 11*(11), 730–755.

Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice* (2nd ed.). New York, NY: Addison-Wesley Professional.

Feldt, K. (2007). *Programming Firefox: Building rich Internet applications with XUL*. Sebastopol, CA: O'Reilly Media, Inc.

Garcia, P. (2011). Command and control rapid prototyping continuum (C2RPC): The framework for achieving a new C2 strategy. In *Proceedings of the 16th International Command and Control Research and Technology Symposium* (ICCRTS), Paper-033, Fairfax, VA.

Gerschefske, K., & Witmer, J. (2012). Wired widgets: Agile visualization of space situational awareness, *Advanced Maui Optical and Space Surveillance Technologies Conference* (AMOS '12). Retrieved from http://www.amostech.com/TechnicalPapers/2012/POSTER/GERSCHEFSKE.pdf

Gizzi, N. (2011). Command and control rapid prototyping continuum (C2RPC) transition: Bridging the valley of death. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (Vol. 1), Naval Postgraduate School, Monterey, CA.

Gorlick, M. M., Strasser, K., & Taylor, R. N. (2012). COAST: An architectural style for decentralized on-demand tailored services. In *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture* (pp. 71–80).

Lopes, C. V. (2011, Sept–Oct). Hypergrid: Architecture and protocol for virtual world interoperability. *IEEE Internet Computing, 15*(5), 22–29.

Maxwell, D. (2012, November 26). Military grid hits performance goals. *Hypergrid Business*. Retrieved from http://www.hypergridbusiness.com/2012/11/military-grid-hits-performance-goals/

Meyers, B. C., & Oberndorf, P. (2001). *Managing software acquisition: Open systems and COTS products*. Addison-Wesley Professional.

Military Open Simulator Enterprise Strategy (MOSES). (2013). Retrieved from http://brokentablet.arl.army.mil/

Reed, H., Benito, P., Collens, J., & Stein, F. (2012). Supporting agile C2 with an agile and adaptive IT ecosystem, in *Proceedings of the 17th International Command and Control Research and Technology Symposium (ICCRTS)*, Paper-044, Fairfax, VA.

Ovaska, P., Rossi, M., & Marttiin, P. (2003). Architecture as a coordination tool in multi-site software development. *Software Process—Improvement and Practice, 8*(4), 233–247.

Ozone Widget Framework. (2013). *Ozone Widget Framework readme*. Retrieved from https://github.com/ozoneplatform/owf/blob/master/README.md

Scacchi, W., & Alspaugh, T. (2012a). Addressing the challenges in the acquisition of secure systems with open architectures. In *Proceedings of the Ninth Annual Acquisition Research Symposium* (Vol. 1), Naval Postgraduate School, Monterey, CA.

Scacchi, W., & Alspaugh, T. (2012b). Understanding the role of licenses and evolution in open architecture software ecosystems. *Journal of Systems and Software, 85*(7), 1479–1494.

Scacchi, W., & Alspaugh, T. (2012c). Developing secure systems using open architectures with open source and closed source components. In *Proceedings of the Eighth IFIP International Conference on Open Source Systems*, *IFIP Advances in Information and Communications Technology* (Vol. 378, 144–159), Hammamet, Tunisia.

Scacchi, W., Brown, C., & Nies, K. (2012). Exploring the potential of virtual worlds for decentralized command and control. In *Proceedings of the 17th International Command and Control Research and Technology Symposium (ICCRTS)*, Paper-096, Fairfax, VA.

Soylu, A., Wild, F., Mödritscher, F., Desmet, P., Verlinde, S., & De Causmaecker, P. (2011). Mashups and widget orchestration. In *Proceedings of the International Conference on Management of Emergent Digital EcoSystems* (MEDES ’11) (pp. 226–234).

Taylor, R. N., Medvidovich, N., & Dashofy, E. (2009). *Software architecture: Foundations, theory, and practice*, New York, NY: Wiley.

Xen.org. (2012). *What is the Xen Hypervisor?* Retrieved from http://xen.org/products/xenhyp.html

THIS PAGE INTENTIONALLY LEFT BLANK

# Ongoing Software Development Without Classical Requirements

## Abstract

Many prominent open source software (OSS) development projects produce systems without overt requirements artifacts or processes, contrary to expectations resulting from classical software development experience and research, and a growing number of critical software systems are evolved and sustained in this way yet provide quality and rich functional capabilities to users and integrators that accept them without question. We examine data from several OSS projects to investigate this conundrum and discuss the results of research into OSS outcomes that sheds light on the consequences of this approach to software requirements in terms of risk of development failure and quality of the resulting system.

**Keywords:** open source software; open source requirements; provisionments.

## Introduction

In 2002, one of us (Scacchi) published a study of requirements practices and artifacts in four open source software (OSS) development communities (Scacchi, 2002). This was the first systematic study to show that OSS system and development processes do not rely on what may be termed classical requirements artifacts and processes, namely those involving problem-space requirements in a document or repository evaluated for completeness and internal and external consistency. Others have since reported similar results (German, 2003; Noll, 2008; Noll & Liu, 2010). Yet there are successful, ongoing OSS projects with users numbered in the millions, and hundreds of OSS systems relied on as critical infrastructure, such as GNU/Linux, the Apache HTTP server, the Mozilla Firefox Web browser, the PostgreSQL database system, and the Eclipse development platform to name a few (Stallman, 2007; Mockus, Fielding, & Herbsleb, 2002; PostgreSQL.org, 2013; Des Rivières & Wiegand, 2004).

From the point of view of a classically trained software developer and requirements practitioner and researcher such as the other of us (Alspaugh), this is unexpected. The broad consensus among software experts and researchers over recent decades has been that devoting appropriate attention to requirements processes and artifacts is essential to project success (Brooks, 1975; Gause & Weinberg, 1989; Jackson, 1995; Lamsweerde, 2009; Sommerville, 2004; van Vliet, 2000) and that failure to do so risks undesirable outcomes such as

- a product that fails to meet stakeholder needs,

- a product that does not exhibit necessary levels of reliability, evolvability, or other software qualities,
- schedule slips and budget overruns, or
- in extreme cases failure to produce any product at all.

How can it be that OSS development produces good software?

In the remainder of this chapter, we explore this conundrum. We present a motivating example, Brooks' thoughts on the success of Linux, and elaborate what we mean by "classical requirements artifacts and processes", hereafter abbreviated as *Classical Requirements*, before describing our study and examining the OSS artifacts and processes that appear to serve in the place of Classical Requirements, using data from our previous work and work reported by others. We find that the overwhelming majority of requirements-like artifacts identified by ourselves and others may be characterized as what we term *provisionments*, which state features or qualities in terms of the attributes provided by an existing software version, a competing product, or a prototype produced by a developer advocating the change it embodies.

The processes involving these artifacts resemble or in some cases are indistinguishable from the bug reporting, tracking, and response processes found in closed source software (CSS) development.

We discuss several contexts in which provisionments appear common and are arguably appropriate: OSS of course, software game mods, and open architecture software ecosystems.

Finally, we place our work in the context of related work, discuss several questions of interest, and conclude the paper.

## A Motivating Example: Brooks on Linux

In reflecting on Raymond's description (Raymond, 2001) of the open source process producing Linux, Brooks observes of this "marvelously functional and robust operating system" that "for Linux a functional specification already existed: Unix" (Brooks, 2010). This is a curious statement, since the development of Unix itself displayed characteristics of OSS development including

- software developed for the developers' own use rather than for an external client and users,
- a strong emphasis on extensibility, and
- no overt requirements artifacts or process preceding development.

Saying that Unix (specifically the Unix kernel) provided the requirements for Linux does not explain the problem; it merely moves it from Linux to Unix. The Unix kernel is marvelously functional and robust, too; was it developed using a functional specification or other Classical Requirements?

If so, supporting evidence is in short supply. Ken Thompson wrote the initial version of Unix in four weeks in the summer of 1969, yet the first edition of the Unix manual was dated  November 3, 1971 (Salus, 1994). Salus noted that “the only way you could learn [the Unix system] was to sit down with one of the authors and ask questions” (Salus, 1994). Ritchie recalled that in 1969, “Thompson, R. H. Canaday, and Ritchie developed, on blackboards and scribbled notes, the basic design of a file system that was later to become the heart of Unix,” not the requirements, but the design (Ritchie, 1984). We have searched the writings of the creators of Unix and researchers reporting on it for Classical Requirements without finding evidence of it. It appears that it is indeed possible to produce a marvelously functional and robust operating system without the aid of a functional specification or other Classical Requirements.

Brooks (2010) further noted, as we and others have, that OSS development works because the developers are users, saying “The whole requirements determination is implicit, hence finessed.” He found no contradiction in ongoing development without Classical Requirements once initial development is successfully complete.

### Classical Artifacts and Processes

Researchers and practitioners have developed many types of requirements artifacts and many requirements processes. We do not consider any of them in detail here. Instead we focus on three characteristics shared by nearly every such approach with which we are familiar:

- a requirements document or central requirements repository, defining the system requirements and providing a criterion for whether a particular candidate requirement is or is not a requirement for the system;
- requirements that are preferentially described in terms of the problem space rather than the solution space; and
- requirements processes for examining the requirements document/repository for completeness, internal consistency, and external consistency with the domain and stakeholder needs.

These characteristics define what we term in this paper *Classical Requirements.*

We focus on these characteristics because they figure prominently in many influential requirements approaches and in the requirements practices of working CSS developers we have known or interviewed, and because convincing arguments have been made from them to project success and product quality (Brooks, 1975; Boehm, 1976; Gause & Weinberg, 1989; Jackson, 1995; Lamsweerde, 2009; Sommerville, 2004; van Vliet, 2000).

Brooks (1987) famously said, “The hardest single part of building a software system is deciding precisely what to build. … No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.”

Boehm (1976) asserted, supported by data, “Clearly, it pays to invest effort in finding requirements errors early and correcting them in, say, 1 man-hour rather than waiting to find the error during operations and having to spend 100 man-hours correcting it.”

Lamsweerde (2009) characterized requirements errors as “numerous and persistent” and as the most expensive and dangerous of software errors. Gause and Weinberg (1989) noted, “Obviously, requirements are important because if you don’t know what you want, or don’t communicate what you want, you reduce your chances of getting what you want.”

The particular form of the requirements is not material to our work. We note that the prominence and importance of particular requirements artifacts and processes often vary depending on the type of system. Not all are appropriate for development of every system, but many situations can benefit from an appropriately chosen selection of them. Some (overlapping) types and corresponding artifact or process choices might be

- embedded systems, in which software is a component of a larger hardware system—a state-based specification;
- real-time systems that must meet specific often-inflexible timing constraints—a temporal logic specification;
- critical or high-assurance systems, for which what is required and what is acceptable must be determined with precision and the cost of failure is high—a model-checkable specification and validation by stakeholders;
- systems that interact significantly with other automated systems—a formalized specification checked for consistency and completeness;
- systems that play a role in specific organizational processes—stakeholder analysis;

- systems that address novel problems or address problems in a novel way—processes that encourage exploration of the problem space.

We note that the use of Classical Requirements in these situations and others may be connected to the typical CSS context in which

- the system is produced by a development group for a client outside that group,
- most or all of the system’s expected users are also outside that group,
- the developers may or may not have expertise in the problem domain, and
- the system is developed against a budget and a schedule.

The requirements state the expectations and commitments of the client on the one hand and the development group on the other. The client balances the benefits of the specific proposed system against the cost of developing it and the wait until it is ready. The development group evaluates whether the budget, resources, and schedule are appropriate for the work involved. The two sides explore, negotiate, and (ideally) agree on a set of requirements. Both sides can then make plans based on specific criteria for acceptance.

## Method

### Research Questions and Metrics

Our goal is to address the apparent conundrum of OSS development (OSSD) that does not use Classical Requirements yet successfully produces high-quality software. We apply the Goal Question Metric approach (Basili, Caldiera, & Rombach, 1994) to produce a measurement model operationalizing our goal into research questions, and associating each question with data that can be evaluated (Figure 1).

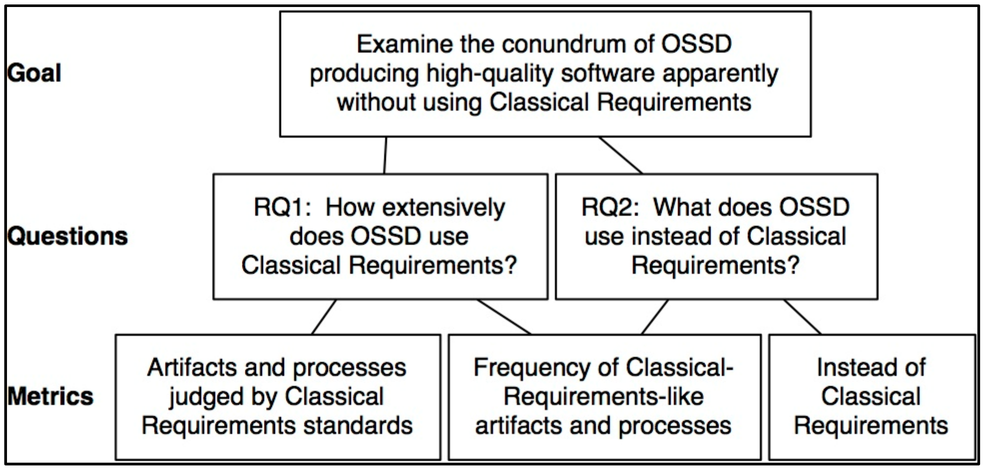

Goal
Examine the conundrum of OSSD producing high-quality software apparently without using Classical Requirements
Questions
RQ1: How extensively does OSSD use Classical Requirements?
RQ2: What does OSSD use instead of Classical Requirements?
Metrics
Artifacts and processes judged by Classical Requirements standards
Frequency of Classical-Requirements-like artifacts and processes
Instead of Classical Requirements

**Figure 1. Goal Question Metric Model**

- (RQ1) To what extent do OSS projects in fact use Classical Requirements?
- (RQ2) Where OSS projects do not use Classical Requirements, what artifacts and processes are used instead, if any?

## Sources of Data

We address RQ1 and RQ2 using data and results from our previous work (Scacchi, 2002, 2009) and from other published research on requirements in OSSD. For an introductory study we find this appropriate, in place of collection of a new set of data. A first step is to identify such research; there is not much. We used work by Noll and Liu (Noll, 2007, 2008; Noll & Liu, 2010) which provides both analysis and some raw data, and work by German (2003) providing analysis only. We also examined the data we found while investigating Brooks' statement that Unix provided Linux's function specification, using it primarily to cross-check, where possible, conclusions we drew from the other data sets. In some cases we followed up on specific data items and examined them in the original context. In a few cases we extended the data with newly-collected data, as for example, that shown in Figure 2.

## Validity

In this subsection we discuss the internal and external validity of the study and threats to its validity.

1. *Internal validity:* Internal validity is the soundness of the relationships within a study. Our study examined data and analysis from different researchers, then merged them in order to apply our metrics. We examined original data where possible in order to apply metrics more uniformly. We looked first for overt Classical Requirements, then for

requirements-like artifacts and processes, and finally for artifacts and processes that appeared to be used in place of requirements. In order to systematize our study, we coded and categorized each such instance, following standard qualitative practice (Creswell, 2003).

2. *External validity:* External validity is the degree to which the results from the study can be generalized. Identification of successful OSS systems without overt Classical Requirements provides an existence proof that software can be successfully developed without it. Other results are more difficult to generalize reliably; for example, the study cannot provide strong support for a hypothesis that Classical Requirements does not contribute to reducing the risk of project failure, nor to increasing the probability that stakeholders will be satisfied. The study also does not provide strong support for hypotheses on the incorporation of OSS development approaches into CSS projects, as our study examines only OSSD data and analyses; these are intriguing and investigation of them remains as future work.
3. *Threats to validity:* We examined every study we found that addressed OSSD requirements, eliminating any possibility of selection bias; however, the number of such studies is quite small (five), making it more difficult to generalize our results and increasing the possibility that other OSSD projects do not fit our conclusions. Other practitioners and researchers might apply different standards, for example with a broader or stricter definition of which instances qualify as Classical Requirements. We minimized this by defining Classical Requirements explicitly and in abstract terms. This threat affects only RQ1.

## OSS Artifacts and Processes

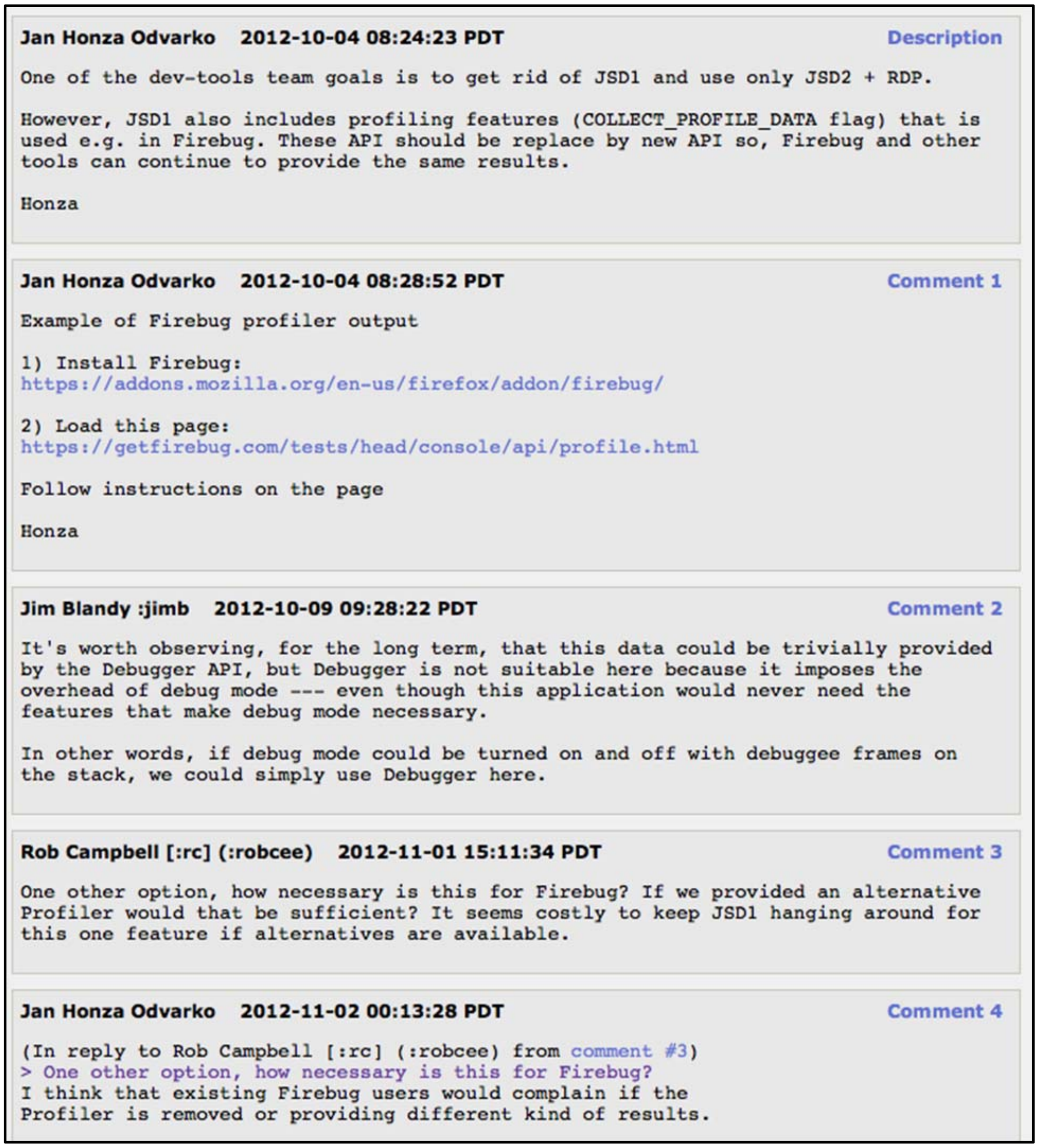
### Requirements-Like Artifacts and Processes

We present several examples of specific requirements-like artifacts and processes we identified in our study. Perhaps the most common requirement-like OSS artifacts are isolated feature requests or bug reports submitted to tracking systems like Bugzilla (Figure 2), and discussed there or on email lists or electronic bulletin boards. An example is this proposal for OpenEMR (Noll & Liu, 2010):

> You could add a link to the existing superbill page which would open a new browser window/tab with a printable version that meets your criteria. This way, you could leverage existing code and probably not have to add a table. I am thinking of something similar to printable links elsewhere in the program, like in reports and patient report.

A second example is shown in Figure 2. Here a Firefox feature request is being discussed, in conjunction with possible changes to the implementation and architecture. Comment 4 may be taken as stating a requirement that Firefox provide the Profiler, specifically, and more generally that Firefox provide a specific kind of results (those that the Profiler currently provided, we infer). This fairly explicit requirement is stated in solution-space terms (what Profiler provides) rather than the corresponding problem-space terms; of course, this is probably considerably more compact. The discussion is focused on architecture and implementation issues involved in the requirement. Other requirements are considered only indirectly if at all, for example if the goal of replacing JSD1 with JSD2 + RDP is taken to imply a here-unstated software quality requirement.

A third example is tabbed browsing, a Web browser feature little known not so many years ago, but now so nearly universal that the name *tabbed browsing* has become a token representing a complex of properties and user stories now assumed to be obvious and requiring no explanation. Mozilla/Firefox tabbed browsing appears to have first been proposed in a one-sentence scenario of use ("One thing that I would really want to see is the ability to open a link in the new window in background …") posted to a Mozilla newsgroup, which was immediately followed by a post beginning "Have you tried tabbed browsing [in the Opera web browser]?" (Noll, 2007). Both these are provisionments; the first cites current system behavior and describes a difference from it, while the second cites another system that exhibits the behavior referred to.



```
Jan Honza Odvarko    2012-10-04 08:24:23 PDT                    Description

One of the dev-tools team goals is to get rid of JSD1 and use only JSD2 + RDP.

However, JSD1 also includes profiling features (COLLECT_PROFILE_DATA flag) that is
used e.g. in Firebug. These API should be replace by new API so, Firebug and other
tools can continue to provide the same results.

Honza

Jan Honza Odvarko    2012-10-04 08:28:52 PDT                    Comment 1

Example of Firebug profiler output

1) Install Firebug:
https://addons.mozilla.org/en-us/firefox/addon/firebug/

2) Load this page:
https://getfirebug.com/tests/head/console/api/profile.html

Follow instructions on the page

Honza

Jim Blandy :jimb    2012-10-09 09:28:22 PDT                     Comment 2

It's worth observing, for the long term, that this data could be trivially provided
by the Debugger API, but Debugger is not suitable here because it imposes the
overhead of debug mode --- even though this application would never need the
features that make debug mode necessary.

In other words, if debug mode could be turned on and off with debuggee frames on
the stack, we could simply use Debugger here.

Rob Campbell [:rc] (:robcee)    2012-11-01 15:11:34 PDT         Comment 3

One other option, how necessary is this for Firebug? If we provided an alternative
Profiler would that be sufficient? It seems costly to keep JSD1 hanging around for
this one feature if alternatives are available.

Jan Honza Odvarko    2012-11-02 00:13:28 PDT                    Comment 4

(In reply to Rob Campbell [:rc] (:robcee) from comment #3)
> One other option, how necessary is this for Firebug?
I think that existing Firebug users would complain if the
Profiler is removed or providing different kind of results.
```

**Figure 2. Discussion of a Feature Request in the Firefox Bugzilla, “Bug 797876—Introduce New API for JS Content Profiling”**

Each feature request or bug report can be taken to imply a requirement, but in themselves they rarely constitute a Classical Requirements artifact. In the examples listed above, as for most requirements-like artifacts we identified, the artifacts are neither integrated into a central requirements document/repository, described in terms of the problem, nor being examined in the context of other requirements. Our study indicates that the OSS projects in question do not use Classical Requirements.

## An OSS Requirements Document

Our data sources included one example identified as a requirements document: “Firefox2/Requirements” (Mozilla Foundation, 2006), discussed by Noll (Noll, 2008). The document is interesting to us in two ways.

First, our examination of the document found the items are expressed in general rather than specific terms, as in this representative example “[The system] will be optimized and tuned for general web browsing use cases,” with the specifics no doubt proposed, discussed, and agreed on through project mailing lists and discussion boards as in the examples in the previous section. We also note that all but one are stated as a difference from the previous Firefox version, using phrases such as “will update” and “will improve,” in other words as provisionments.

Second, and perhaps more significant, this is the only presumptive requirements document or repository our research identified in our own searches and in related work on requirements in OSS. While its existence indicates that OSS development can tend toward Classical Requirements, its apparent uniqueness highlights our general finding that OSS development does not make use of Classical Requirements.

## Provisionments

As stated previously, a *provisionment* is a statement of features or qualities in terms of the attributes provided by an existing software version, a competing product, or a prototype produced by a developer advocating the change it embodies. Most provisionments we encountered only suggest or hint at the behavior or quality in question; the expectation seems to be that the audience for the provisionment is either already familiar with what is intended or will play with the cited system and see the behavior or quality in question firsthand.

In our study, we saw provisionments being used for requirements or requirements-like artifacts in two ways, either directly as a specification of behavior or quality, or as a starting point in a specification of behavior or quality differing in stated ways from that expressed by the provisionment.

Later in this chapter, we provide examples of both types. Firefox Bugzilla comment 4 in Figure 2 (“I think that existing Firebug users would complain if the Profiler is removed or providing [*sic*] different kind of results”) uses a provisionment directly (though stated in the negative), while the OpenEMR proposal uses a provisionment (“the existing superbill page”) indirectly as a starting point for a difference (“You could add …”).

A provisionment is distinct from a feature, a quality, a bug report, and similar entities in that each of those is *something to be expressed*, while a provisionment is *a way of expressing something*. In our study, we found many feature requests and bug reports expressed using provisionments; OSS project archives appear to teem with feature requests and bug reports, and the majority we examined were expressed using provisionments. Statements of qualities were much less common but were also often expressed with provisionments.

## Some Example Contexts

We discuss three contexts highlighting the interplay between requirements, provisionments, and architecture: open source software, here discussed at greater length; software games, some of which are themselves OSS and many of which support modifications that exhibit OSS characteristics, whether the underlying game is OSS or not, and are described using provisionments; and OA systems of complex components, for which provisionments mediated by architectural configurations play prominent roles.

### Open Source Software

OSS requirements, to the extent that they can be identified, tend to be distributed across space, time, people, and the artifacts that interlink them. OSS requirements are thus decentralized—that is, they are decentralized requirements that co-exist and co-evolve within different artifacts, online conversations, and repositories, as well as within the continually emerging interactions and collective actions of OSSD project participants and surrounding project social world. To be clear, decentralized requirements are not the same as the (centralized) requirements for decentralized systems or system development efforts. Traditional software engineering and system development projects assume that their requirements can be elicited, captured, analyzed, and managed as centrally controlled resources (or documentation artifacts) within a centralized administrative authority that adheres to contractual requirements and employs a centralized requirements artifact repository—that is, centralized requirements. In this way as in others, OSSD projects represent an alternative paradigm to that long advocated by software engineering and software requirements engineering community (Scacchi, 2009).

By the standards of classical software development and requirements practice, OSS requirements and processes are not satisfactory. Requirements are expressed indirectly at best; they are scattered across mailing lists, discussion boards, and bug trackers rather than collected in one place; they appear to be integrated only in the implementation of the system they refer to; they are almost universally stated in solution terms, not problem terms; once stated and discussed, they rarely appear to be referred to.

An RE researcher or practitioner might well look at dispersed statements such as these and simply conclude that requirements were for practical purposes absent by any reasonable or ordinary standard; if such decentralized, indirect requirements were used for a classical software development project, it would be judged to be at high risk of failure.

One would think therefore that many open source projects should fail—and they do, in large numbers. About 59% fail according to one study (Wiggins & Crowston, 2010), roughly double the 31% rate at which classical projects are reported to fail according to a 1994 survey (Standish Group, 1994). Of course failure means something different for an OSSD project; there is no concept of over budget or behind schedule, and failed OSSD projects tend to wither away rather than being canceled. Nevertheless, the comparison is startling.

Though most OSSD projects fail to produce a sustained sequence of widely-used software system releases, a substantial number are striking successes. Hundreds of OSSD projects are critical in a number of areas:

- the operation of the World Wide Web (the Firefox and Chrome web browsers and the Apache Web servers and Web services infrastructure),
- interactive software development (Eclipse and NetBeans development environments),
- customer relationship management (SugarCRM),
- database management systems (PostgreSQL, MySQL),
- operating systems (GNU/Linux, Darwin/OSX),
- office communications systems (Asterix),

and many more.

Clearly OSSD processes are capable of producing high quality software systems, despite scanty requirements artifacts and processes. We see the use of provisionments to make statements about the functionality of current and future system versions as one key factor, particularly convenient for an ongoing project producing version after version, each of which is described not in absolute terms but in terms of its differences from the previous one. Others may include developing an (informal) architecture and reasoning about it, in place of developing requirements and reasoning about requirements; using extensibility (see below), developer prototypes, and frequent releases of new system versions to explore the problem space by experimenting with alternative solutions within it; the fact that OSS developers are also users of the systems they develop; and the extensive discussions of system issues and proposals, characteristic of OSSD projects, in online forums that are public and persistently available.

We note that many prominent OSS systems are strongly extensible, with mechanisms by which the core functionality of the system may be extended independently, without affecting the system core. These mechanisms allow end-users to customize their copy of a system to suit their own needs and preferences, and in many cases allow developers to expeditiously prototype candidate provisionments. Examples of extensibility include Unix, supporting the addition of shell scripts, commands, libraries, and device drivers; Firefox, Eclipse, jEdit, and others, supporting the addition of plug-ins; and Firefox and jEdit again, and others, supporting the use of scripting languages. In addition to satisfying the system quality requirement (QR) of extensibility, extension mechanisms can also contribute to the requirement, for project success and continuation, to bring new contributors into the project community. Writing extensions for one's one copy of a system is an easy and appealing first step towards making more substantial contributions to the project that produces the system.

Extensibility and several other quality requirements will be seen to play important roles in games and OA systems too.

Viewing OSSD from a classical RE standpoint, we still note some concerns. Classical RE has approaches for identifying relevant stakeholders, and we see no corresponding practice in OSSD. We are concerned that OSSD projects will tend not to identify stakeholder roles in which the stakeholders are not developers and (for whatever reason) not motivated to come forward and contribute. We are also concerned about the effectiveness of OSSD in exploring the problem space, as opposed to the solution space. If such exploration is occurring, it is doing so inconspicuously.

We also do not claim that developers can easily see into their own goals and needs; they are only human, after all. We note only that what corresponds to elicitation may be more straightforward since the communication step vanishes.

### Software Game Mods

Many software games are extensible and thus can be modified by their users to produce new games, ranging from simple modifications obviously similar to the host game to others almost unrecognizable as related to their hosts.

User-modified computer games, hereafter referred to as *game mods*, are a leading form of user-led innovation in game design and game play experience. Game mods, modding practices, and modders are in many ways quite similar to their counterparts in the world of OSS development, even though they often seem isolated to those unaware of game software development. Modding is increasingly a part of mainstream technology development culture and practice, and especially so for games. Modders are players of the games they reconfigure, just as OSS developers are users of the systems they develop. There is no systematic distinction between developers and users in these communities, except for the many users/players that contribute little beyond their usage and their demand for more such systems. Modding and OSSD projects are in many ways comparable experiments to prototype alternative visions of what innovative systems might be in the near future, and so both are widely embraced and practiced as a means for learning about new technologies, new system capabilities, new working relationships with potentially unfamiliar teammates from other cultures, and more (Scacchi, 2007).

Game conversion mods are perhaps the most common form of game mods. Most such conversions are partial, in that they add or modify in-game characters, game resources such as weapons, potions, or spells, play levels, zones, landscapes, game rules, or play mechanics. In these cases, the conversion can often best be described in terms of provisionments of the host game. More ambitious modders go as far as to accomplish either total conversions that create entirely new games from existing games of a kind that are not easily determined from the originating game, or even parodies that implicitly or explicitly spoof the content or play experience of one or more other games via reproduction and transformation.

One of the most widely distributed and played total game conversions is the *Counter-Strike* (*CS*) mod of the *Half-Life* first-person action game from Valve Software. The *CS* mod attracted millions of players preferring to play it over the original *Half-Life* game. Other modders began to further convert the *CS* mod in part or fully, to the point that Valve Software modified its game development and distribution business model to embrace game modding as part of the game play experience provided by the *Half-Life* product family. Valve has since marketed a number of *CS* variants. As of 2011, Valve Software had sold over 25 million copies of *CS* and its descendants (Makuch, 2011).

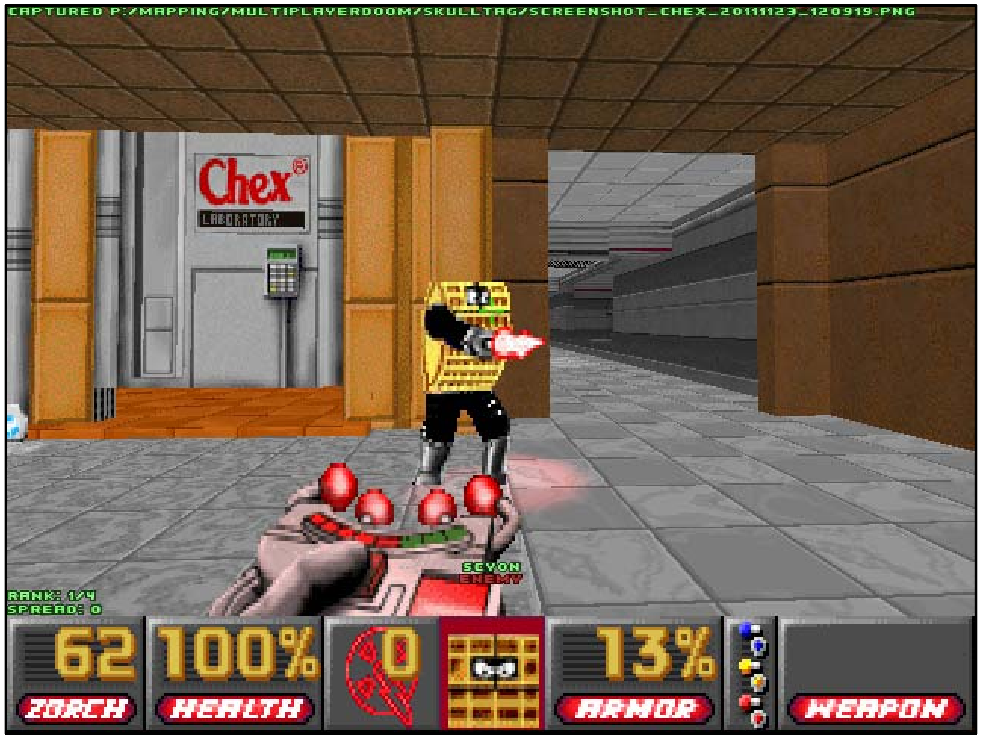

CAPTURED P:/MAPPING/MULTIPLAYERDOOM/SKULLTAG/SCREENSHOT_CHEX_20111123_120919.PNG
Chex
LABORATORY
SCYON
ENEMY
RANK: 1/4
SPREAD: 0
62
ZORCH
100%
HEALTH
0
13%
ARMOR
WEAPON

**Figure 3. A Screenshot of Chex Quest, a Nonviolent Mod of Doom (Image Courtesy of User Vulpis Alba)**

Other player-modders have produced meta-mods, or mods that can themselves be modded, such as *Garry's Mod* of *Half-Life 2*. *Garry's Mod* has evolved into a modding toolkit used in hundreds of game conversions and producing inventive game play mechanics. Game conversions can also exhibit innovations in game design and re-purposing. The game *Chex Quest* is a conversion of the first-person shooter game *Doom* into a non-violent game distributed in Chex cereal boxes and targeted to young people and gamers (Figure 3).

Extensibility to support the creation of mods has become a necessary feature for a successful game.

## Open Architecture Software Ecosystems

As we note in our previous work (Alspaugh, Asuncion, & Scacchi, 2009, 2013; Alspaugh, Scacchi, & Asuncion, 2010; Scacchi & Alspaugh, 2012b), a substantial number of development organizations have adopted a strategy in which a software-intensive system is developed with an open architecture (OA; Oreizy, 2000), integrating components that may be OSS or proprietary with open application programming interfaces (APIs). Such systems evolve not only through the evolution of their individual components but also through replacement of one component by another, possibly from a different producer or under a different license. With this approach, the development organization becomes an integrator of components largely produced elsewhere and interconnected through open APIs, with shim code added as necessary to achieve the desired result. This approach allows development of large systems of complex components, with relatively little coding needed. Requirements artifacts and processes are not prominent here. Instead, we see a prototyping process and a system described in terms of provisionments rather than requirements.

One reason that reasoning with provisionments is appealing for OA systems is that the integrator cannot choose arbitrary functional capabilities. Instead, there are a limited number of alternative components to select among, and one must simply take what is available. As the components evolve the same situation recurs, in that the functional capabilities may change from version to version, and the integrator must work with what is available. The most straightforward approach is simply to reason based on what the selected components provide.

A second reason is that individual components such as Firefox do not come with Classical Requirements that could be used to reason about requirements for the overall system.

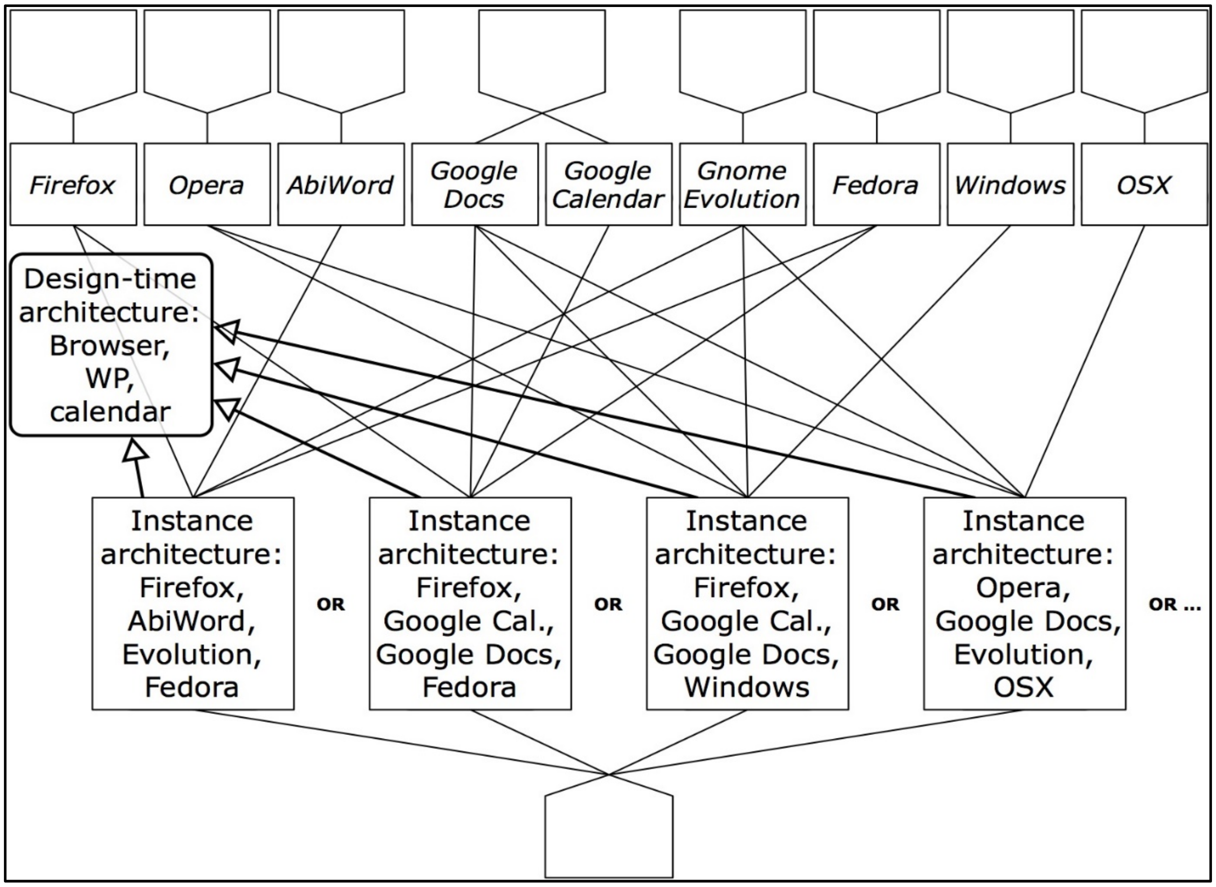

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
OR
Instance architecture: Firefox, Google Cal., Google Docs, Fedora
OR
Instance architecture: Firefox, Google Cal., Google Docs, Windows
OR
Instance architecture: Opera, Google Docs, Evolution, OSX
OR ...

**Figure 4. Ecosystem From Which Instantiations of the System Architecture Can Be Drawn**

The possible components that can be incorporated into a system define an ecosystem for it. Figure 4 sketches a potential ecosystem for a system composed of a Web browser, word processor, email and calendar component, and any scripts and shim code the integrator produces to knit them all together and achieve the desired functionality. If we hypothetically consider the requirements of the composed system, we note that the requirements would necessarily be decentralized, since whatever requirements process we used for the overall system would be independent of that used for each individual component. If we were able to get requirements for each component (which in general is not possible) and integrate them to arrive at requirements for this version of the overall system, this central requirements artifact would last only until the next component version was released, sending the situation back to decentralized requirements.

In practice, integrators appear to follow the lead of the developers of the OSS components and work with provisionments. The acceleration of evolution caused by integrating the independent supply chains for the components currently selected is driving a need to understand decentralized requirements and reason in terms of decentralized provisionments.

## Related Work

### Requirements in Open Source Development

Scacchi (2002) was the first to systematically observe and posit the idea that OSS system and development processes do not rely on producing and review of formal functional requirements documents. Instead, OSS development projects commonly rely on “software informalisms,” no matter what the application domain, nor who the developers may be. Such informalisms are rendered within online artifacts like bug reports, messages in a discussion forum, online chat transcripts, and so forth that developers use to communicate their interests about different aspects of a system, its development, its user experience, or its need to evolve in some way. He found that OSS requirements often were described after the functionality they prescribe had already been implemented and found to be viable or practical—requirements after the fact. By 2009, Scacchi had identified a set of more than 20 different types of informalisms in use across different open source development (OSSD) projects, such that a given project might routinely use five to 10 informalisms, with different projects utilizing different mixes of software informalisms so that no specific set seems to dominate (Scacchi, 2009). The informalisms identified were (a) project email; (b) discussion forums, electronic bulletin boards, and group blogs; (c) news postings; (d) instant messaging; (e) project digests summarizing (a)–(d); (f) usage scenarios as linked Web pages or screenshots; (g) how-to guides; (h) to-do lists; (i) Frequently Asked Questions lists; (j) project Wikis; (k) traditional system documentation; (l) external publications; (m) project licenses; (n) open software architecture diagrams; (o) intra-application functionality in scripting languages; (p) externally developed software modules (“plug-ins”); (q) software modules reused from other OSS projects; (r) project Web sites or portals; (s) project source code Web directories; (t) project repositories such as CVS; (u) bug reports; and (v) issue tracking databases such as Bugzilla. Provisionments may be found in many of these informalisms—especially (a–e), (u), and (v)—but the category of provisionments is orthogonal to them and, we believe, significant in itself.

German (2003) described five sources for requirements for the GNOME project, based on his experience as a contributor. He termed them *vision* (a leader proposes a list of requirements), *reference application* (an outside system is to be imitated), *asserted requirement* (arising from discussions among contributors), *prototype* (an implementation illustrating a proposed feature to be discussed), and *post-hoc requirement* (like a prototype, but offered as a ready-to-integrate implementation of a feature the contributor desires). Provisionments are most closely involved with German’s prototypes and post-hoc requirements.

Noll (2008) examined the published requirements document for the Web browser version Firefox 2.0, identifying where each of the 14 items was first mentioned, how it was implemented, and why each was initially proposed. Eight were asserted by developers from their personal experience or knowledge of user needs, three were requested by users, and one was driven by a feature in competing browsers. This highlights that although OSS developers are themselves users, non-developer users also play a role in OSS evolution.

Noll and Liu (2010) also examined requirements for the OpenEMR electronic medical records project, finding comparable proportions contributed by developers versus users. Each feature was briefly discussed in the project’s online developers forum, which they characterized as requirements validation and agreement. We found the OpenEMR requirements or features to be more difficult to classify, for example “Support for deleting immunizations,” and hypothesize that each acts as a token for the corresponding forum discussion.

### Requirements and Architecture

The close relationship between requirements and architecture suggests that the affordances provided by requirements in classical development may somehow be provided through architectural means in OSSD.

Nuseibeh (2001) proposed the Twin Peaks model as an expression of the interrelation of requirements and architecture: Problem concerns and solution concerns cannot in general be addressed in sequence, rather needing to be addressed concurrently. The model conveys a back-and-forth alternation treating both requirements and architecture in increasing detail.

De Boer and van Vliet (2009) argued that the traditional distinction between requirements and architecture is misguided, and that there is no fundamental difference between them, saying “architecturally significant requirements [ASRs] are in fact architectural design decisions [ADDs], and vice versa.” Both are optative statements characterizing what is desired, and by their nature earlier optative statements constrain what later optative statements can be made.

Diallo, Sim, and Alspaugh (2007) found that of systems with published development artifacts, only toy systems for textbooks have both complete requirements and a complete architecture. Of the remainder, roughly half had a complete architecture, another quarter had complete requirements, and the remainder had neither. We believe this occurs because requirements and architecture are to a certain degree redundant, so that developers have no need to develop both fully.

All this work suggests that if expected OSS requirements artifacts or processes do not appear to be present, the purposes of those artifacts and processes may be achieved through architectural means.

## Discussion

### Are OSS Requirements “Good”?

This is a fascinating question to which we have no definitive answer.

In one sense, the answer is “most definitely not.” The previous career of one of us (Alspaugh) included work as a developer, team lead, manager, and consultant occasionally called in to help struggling development projects. In each case, the struggles could usefully be ascribed to problems with the project’s requirements artifacts and processes, in that attacking those problems brought the projects in each case onto a path that could (and usually did) lead to success, and the OSS requirements–like artifacts and processes we examined evoke the problematic ones of those projects.

However, the OSS data we examined in this study was not from troubled projects but from flourishing ones. We conclude that at least some of the work that Classical Requirements accomplishes is being done in another domain with processes appropriate to that domain; our hypothesis, potentially supported by some of the data we examined, is that some of it is being done in the software architecture domain, through processes that are more what we would expect though here again the artifacts do not appear to be overt.

We note again that CSS bug reports and feature requests and the processes for managing them look much like those for OSS.

### Centralized vs. Decentralized

Rather than a single central requirements or provisionments repository or document, updated as necessary, OSS projects almost universally appear to use email threads, electronic bulletin boards, and similar sequences of archived interactions as a record of them (and of virtually everything else, it appears).

This choice prevents overall consideration and analysis of the provisionments as a whole. However, it may support a deeper goal for OSSD projects: creating and sustaining a community of contributors. The ongoing conversation, archived online so potential contributors can dip into it to see if interests them, provides an ongoing sequence of nudges to participate and a continuing reinforcement of community membership to those who do participate. This may be more valuable and fundamental than any incremental benefits likely to accrue from unifying the information into a single document.

### Is OSSD Efficient?

There do not appear to be data on this question yet. It is not clear that successful OSS projects produce results as expeditiously or more so than CSS projects do; they may well be slower in calendar time or take more person-months. Certainly the importance of schedules and budgets in CSS could drive more efficient development. Brooks noted that one would expect communication to be a more serious bottleneck for OSS than for CSS (Brooks, 2010), though we note this may be ameliorated by the reduction or elimination of communication between developers and stakeholders, since OSS developers are themselves users and stakeholders.

### Would OSS Benefit from Classical Requirements?

Perhaps, but the answer is not clear; at this stage, we can only speculate. If the user-developers are identifying stakeholder needs sufficiently well and those needs are addressed sufficiently well by the incremental revisions that appear to characterize OSSD, then probably not. However if the needs would be best addressed by a reconsideration of the problem and a more radical change in the solution, Classical Requirements has advantages to offer.

We note the truism that a new solution to a problem opens the eyes of its users to new problems not previously considered. A product that is evolving at a sufficiently rapid pace (and OSS systems are considered to evolve rapidly) may be obtaining many of the benefits of problem-space requirements processes through solution-space development processes.

### Are Provisionments Advantageous?

We see an increasing trend of rapidly-evolving systems described and reasoned about in terms of whole-system provisionments, or of component provisionments related through the system’s architecture (Alspaugh, Asuncion, & Scacchi, 2009, 2013; Alspaugh, Scacchi, & Asuncion, 2010; Scacchi & Alspaugh, 2012a). This may not only be increasingly typical but also in fact the appropriate approach for reasoning about a stakeholder problem and complex system solution, that is to be implemented by combining complex components. Such an approach manages complexity by reasoning in terms of the capabilities of known (though often themselves complex) components, arranged in architectural configurations in which the capabilities combine to address a problem. It manages ongoing evolution by describing future behavior in terms of differences from past behavior.

### Are Provisionments Limited to OSS?

No, they are not; we have seen them in our work as professional CSS developers, most prominently in bug reports and to a lesser extent in feature requests where they serve the same purposes as in OSS.

Some professional CSS developers with whom we have discussed this research report that the requirements they work with might frequently be more accurately described as provisionments. And as we noted previously, OA system development often appears to be guided by reasoning with provisionments, whether the integrators are an OSS project or a proprietary development group, and with good cause.

As we and many other researchers have noted, there is now far more data available from OSS development projects than there is from CSS projects, to which researchers typically have limited or no access. We recall the challenges we have faced in attempting to get access to proprietary development requirements in order to do research. Based on our results so far, we expect provisionments will be found to be in wide use in OSS development, or even in virtually universal use since they align so naturally with reported OSSD processes. It will be more difficult to assess the degree to which provisionments are used in CSS development, but based on what we have learned, we believe their use is widespread there also.

## Conclusion

In this paper we examined the apparent contradiction between the success of at least some OSS systems and their lack of what may be termed classical requirements artifacts and processes or *Classical Requirements*. Then we listed four research questions. Here we summarize the answers arising from our study and our examination of related work.

*(RQ1) To what extent do OSS projects in fact use Classical Requirements?* In the data we examined, Classical Requirements was almost completely absent. We found requirements-like artifacts and some requirements-like processes, but virtually nothing exhibiting the three characteristics by which we defined Classical Requirements in a previous section.

*(RQ2) Where OSS projects do not use Classical Requirements, what artifacts and processes are used instead, if any?* The most prominent requirements-like artifacts we identified were provisionments, statements of features or qualities in terms of the attributes provided by an existing software version, a competing product, or a prototype produced by a developer advocating the change it embodies. These were ubiquitous in the data we examined. The processes were more difficult to characterize; perhaps the most common requirements-like process we saw was the discussion of provisionments in terms of solution-space issues. We hypothesize

that architectural reasoning and discussion played a role as well, but did not find strong evidence for it; we may have been looking in the wrong places for that.

In summary, OSS's lack of Classical Requirements results in some of the undesirable outcomes predicted by the broad consensus of software experts and researchers, but not all of them. In some contexts the advantages of OSS appear to outweigh this disadvantage. Further research will be needed to obtain more definitive answers and to provide guidance to making the most effective use of OSS development approaches.

## Acknowledgments

This research is supported by grant #N00244-12-1-0067 from the Acquisition Research Program at the Naval Postgraduate School, and by grant #1256593 from the U.S. National Science Foundation. No review, approval, or endorsement is implied.

The authors thank the anonymous reviewers of an earlier version of this paper for their careful and insightful comments.

## References

Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2009). Intellectual property rights requirements for heterogeneously-licensed systems. In *Proceedings of the 17th IEEE International Requirements Engineering Conference (RE '09)* (pp. 24–33).

Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2013). The challenge of heterogeneously licensed systems in open architecture software ecosystems. In Jansen, S., Cusumano, M., & Brinkkemper, S. (Eds.), *Software ecosystems: Analyzing and managing business networks in the software industry*. Edward Elgar Publishing.

Alspaugh, T. A., Scacchi, W., & Asuncion, H. U. (2010). Software licenses in context: The challenge of heterogeneously-licensed systems. *Journal of the Association for Information Systems*, *11*(11), 730–755.

Basili, V. R., Caldiera, G., & Rombach, H. D. (1994). The Goal Question Metric approach. In *Encyclopedia of Software Engineering* (pp. 528–532). John Wiley and Sons.

Boehm, B. (1976). Software engineering. *IEEE Transactions on Computers*, *25*(12), 1126–1241.

Brooks, F. P., Jr. (1975). *The mythical man month: Essays on software engineering* (1st ed.). Addison-Wesley.

Brooks, F. P., Jr. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer, 20*(4), 10–19. Reprinted from IFIP Congress, Dublin, Ireland, 1986.

Brooks, F. P., Jr. (2010). *The design of design: Essays from a computer scientist*. Addison-Wesley.

Creswell, J. W. (2003). *Research design: Qualitative, quantitative, and mixed methods approaches* (2nd ed.). Thousand Oaks, CA: SAGE.

de Boer, R. C., & van Vliet, H. (2009). Controversy corner: On the similarity between requirements and architecture. *Journal of Systems and Software*, *82*(3), 544–550.

Des Rivières, J., & Wiegand, J. (2004). Eclipse: A platform for integrating development tools. *IBM Systems Journal, 43*(2), 371–383.

Diallo, M., Sim, S. E., & Alspaugh, T. A. (2007). Case study, interrupted: The paucity of subject systems that span the requirements-architecture gap. In *First Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech '07)*.

Gause, D. C., & Weinberg, G. M. (1989). *Exploring requirements: Quality before design*. New York, NY: Dorset House.

German, D. M. (2003). GNOME, a case of open source global software development. In *International Workshop on Global Software Development (GSD '03)*.

Jackson, M. (1995). Software requirements and specification: A lexicon of practice, principles and prejudices. Wokingham, England: Addison-Wesley.

Lamsweerde, A. v. (2009). *Requirements engineering: From system goals to UML models to software specifications*. Wiley.

Makuch, E. (2011). Counter-Strike: Global offensive firing up early 2012. Retrieved from http://www.gamespot.com/6328645

Mockus, A., Fielding, R. T., & Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology, 11*(3), 309–346.

Mozilla Foundation. (2006). *Firefox2/Requirements*. Retrieved January 27, 2013, from http://wiki.mozilla.org/Firefox2/Requirements

Noll, J. (2007). Innovation in open source software development: A tale of two features. In Feller, J., Fitzgerald, B., Scacchi, W., & Sillitti, A. (Eds.) *Open* *source development, adoption and innovation: IFIP Working Group 2.13 on open source software* (pp. 109–120). Springer.

Noll, J. (2008). Requirements acquisition in open source development: Firefox 2.0. In Russo, B., Damiani, E., Hissam, S., Lundell, B., & Succi, G. (Eds.). *Open source development, communities and quality (IFIP—The International Federation for Information Processing)* (pp. 69–79). Springer-Verlag.

Noll, J., & Liu, W.-M. (2010). Requirements elicitation in open source software development: A case study. In *Proceedings of the Third International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS '10)* (pp. 35–40).

Nuseibeh, B. (2001). Weaving together requirements and architectures. *IEEE Computer, 34*(3), 115–117.

Oreizy, P. (2000). *Open architecture software: A flexible approach to decentralized software evolution* (Doctoral dissertation). University of California, Irvine. Retrieved from http://www.ics.uci.edu/~peymano/papers/thesis.pdf

PostgreSQL.org. (2013). About. Retrieved from http://www.postgresql.org/about/

Raymond, E. S. (2001). *The cathedral and the bazaar: Musings on Linux and open source by an accidental revolutionary* (Rev. ed.). O'Reilly.

Ritchie, D. (1984). The evolution of the Unix time-sharing system. *AT&T Bell Laboratories Technical Journal, 63*(6), 1577–1593.

Salus, P. H. (1994). *A quarter century of UNIX*. Addison-Wesley.

Scacchi, W. (2002). Understanding the requirements for developing open source software systems. *IEE Proceedings—Software, 149*(1), 24–39.

Scacchi, W. (2007). Free/open source software development: Recent research results and emerging opportunities. In *Proceedings of the Sixth Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)* (pp. 459–468).

Scacchi, W. (2009). Understanding requirements for open source software. In Lyytinen, K., Loucopoulos, P., Mylopoulos, J., & Robinson, B. (Eds.). *Design requirements engineering: A ten-year perspective* (pp. 467–494). Springer-Verlag.

Scacchi, W., & Alspaugh, T. A. (2012a). Designing secure systems based on open architectures with open source and closed source components. In *International Conference on Open Source Systems (OSS 2012).*

Scacchi, W., & Alspaugh, T. A. (2012b). Understanding the role of licenses and evolution in open architecture software ecosystems. *Journal of Systems and Software, 85*(7), 1479–1494.

Sommerville, I. (2004). *Software engineering* (7th ed.). Addison-Wesley.

Stallman, R. (2007). Linux and the GNU system. Retrieved from http://www.Gnu.org/gnu/linux-and-gnu

The Standish Group (1994). *The CHAOS report*. Boston, MA: Author.

Van Vliet, H. (2000). *Software engineering: Principles and practice* (2nd ed.). John Wiley & Sons.

Wiggins, A., & Crowston, K. (2010). Reclassifying success and tragedy in FLOSS projects. In *Proceedings of the Sixth International Conference on Open Source Systems* (pp. 294–307).

# Moving Towards Formalizable Security Licenses

## Abstract

Security policies informally stipulate functional and non-functional requirements for how software systems should be designed and implemented to defend the system against attack. These policies must be both human-readable and computationally enactable. Stakeholders must be able to read and understand them in order to assess whether a policy meets the stakeholder’s needs. Software development environments must be able to compute with them in order to integrate components in ways that continuously assure the compliance with security policy specification. Computational processing of security requirements is particularly important when integrating unrelated components from a variety of software producers, as is increasingly the case. We introduce *security licenses* as a new approach for meeting both these goals. We have used an analogous approach for calculating how the software copyright licenses of open source and proprietary components interact in open architecture systems whose components come from disparate suppliers. The restricted natural language specification of security licenses provides readability, while the formal structures underlying them support computations to identify conflicts and gaps, provide guidance, and assess the results for the integrated system.

## Introduction

Security policies and similar artifacts express a range of security requirements and mechanisms, such as

1. discretionary versus mandatory access control lists;
2. firewalls;
3. multi-level security;
4. authentication (certificate authorities, passwords);
5. cryptographic support (e.g., public key certificates);
6. encapsulation (e.g., virtualization, abstract APIs), hardware isolation schemes (Sun, Wang, Zhang, & Stavrou, 2012), and access control type enforcement capabilities;
7. data content or control signal flow logging/auditing;
8. honey-pots and traps;
9. functionally equivalent but diverse multi-variant software executables (Franz, 2010; Salamat, Jackson, Wagner, Wimmer, & Franz, 2011);

10. security technical information guides (STIGs) for configuring the security parameters for applications (Defense Information Systems Agency, 2011) and operating systems (Smalley, 2012);
11. secure programming practices (Seacord, 2008); and
12. standards for development organization processes and practices (ISO/IEC, 2005).

Some of these are computer processable, and others are easy for stakeholders to read and understand. However, few seem to be both. The processable items are typically at a low level (e.g., user I/O privileges specified by mandatory access control lists) while the easily-understood items are typically at a high level. The reader will also note that these are *software implementation choices* or *software process choices* rather than *system architectural choices* or *security requirements/policy choices*. Between these mechanisms and a workable concept of a comprehensive security policy for a system or its substantial components is a gap, with no obvious way to bridge it.

- There is no common framework or conceptual basis with which to systematically specify, integrate, and evaluate mechanisms in combination, particularly at the software architectural level where diverse components are interconnected into buildable system configurations. It is unclear how the various security mechanisms are related and how one may contribute to or interfere with another.
- Guidance is scant for analysts, architects, and developers who need to decide which security mechanism to use where, when, how, and why; and also for system integrators and administrators who need to decide how to update the selection of mechanisms and their configuration within a system as security needs and policies evolve.

No satisfactory framework exists in which software security policy elements or mechanisms can be assembled into common or reusable patterns that can be designed and combined in a software system architecture to meet specific high-level security policies and requirements.

We believe there is an opportunity to address security challenges throughout the system development and deployment process using *security licenses*.

In our previous work (Alspaugh, Asuncion, & Scacchi, 2009, 2011, 2012; Alspaugh, Scacchi, & Asuncion, 2010; Alspaugh, Scacchi, & Kawai, 2012; Scacchi & Alspaugh, 2012) we showed how software copyright licenses for independently developed system components can be used to guide architectural choices and evaluate rights and obligations for the system as a whole, even when components are governed by different licenses. Using our approach, a system architect can work both down from the top, propagating desired license rights for the system down to individual components to see what license obligations are required to obtain those rights, and up from the bottom, combining license rights and obligations for components and then subsystems into the total rights and obligations for the system. In either direction, our approach identifies conflicts or mismatches among licenses in the architecture.

We propose an analogous approach for security licenses. System analysts and architects choose an appropriate security license or create a new one from desired security rights and acceptable security obligations, assign a candidate security license to each subsystem or component, and calculate the interactions between these choices at every level from an individual component up to the complete system. Of course assigning a particular security license to a component does not guarantee the component's developer will make it satisfy its security obligations, any more than accepting a component under the GNU General Public License (GPL; Free Software Foundation, 2007) guarantees that the system's stakeholders will satisfy all the obligations GPL imposes. But assigning a license (whether security or copyright) to each component records the assumptions being made about that component and its use, and evaluating those licenses in the context of the system's architecture identifies mismatches and conflicts among those assumptions for that architecture's design choices. When the evaluation is automated, as it is in our work (Alspaugh et al., 2011), it forms the foundation for design guidance with respect to the issues raised by the licenses, and a means for combining components with potentially dissimilar licenses in a configuration that can satisfy all of them. This means we can determine the overall interaction and effect of the security mechanisms that are expected to satisfy the obligations and of the security requirements and policies that the rights express.

## Effectiveness, Manageability, Evolvability

Consider the case of the development of an open-architecture (OA) software system that integrates proprietary closed source (executable binaries) and open source code components from a variety of producers who do not coordinate their security activities or run-time capabilities. From the point of view of ensuring security, this is arguably the worst possible case, but it is an increasingly prevalent development model (Alspaugh et al., 2010). The OA approach gives access to a wide selection of complex components of high quality and allows the system to evolve as quickly as its integrators can find appropriate new versions or new components and evolve their architecture and shim code to accommodate them.

Since the producers do not coordinate, they are unlikely to use the same security approaches, and indeed may not even publish what those approaches are.

To control security in the resulting system, each component is enclosed in a *containment vessel* (Scacchi & Alspaugh, 2013) that isolates the component with a hypervisor (Xen Hypervisor Project, 2012) and mediates all communication with the component (method/function calls, data streams) through shim code that monitors and restricts it.

A typical current-day technique (Luo & Du, 2011) for managing security measures is to use low-level implementation mechanisms like *capability lists* to control each user/component's access to resources such as function calls and data updates. Each access is delayed briefly while the monitor checks the access against the accessing component's capability list, then blocked if the component was not granted the capability to access that resource. In our experience, each capability list is a text file listing allowed or forbidden capabilities, managed manually; new capabilities are typically added to the end of the file. In general, there appears to be no formal model supporting relationships among capabilities, thus interactions between capabilities must be identified and managed manually. This becomes extraordinarily tedious once the number of software components and diversity of user roles increases. The text files are detailed, which is a positive aspect, but therefore also long and mind-numbingly tedious, so errors inevitably creep in and are not noticed. All too often, available and viable low-level security capabilities are not utilized because they are so unwieldily, and their interactions are opaque. Because a capability list has no architectural, hierarchical, or recursive structure, managing them is not scalable. High-level security policies and low-level security capability mechanisms just don't mesh well at present, and thus too often preventable security gaps arise and persist.

A more sophisticated approach is possible using a declarative policy language such as Ponder (Damianou, Dulay, Lupu, & Sloman, 2001) or an ontology-based language such as KAoS (Uszok, Bradshaw, Johnson, et al., 2004) that groups capabilities hierarchically, in ontologies (KAoS) or grouped by roles (Ponder). However, they have no provision for organizing capabilities by software components that can be configured into tractably secure system architectures.

## Human Readable, Computer Processable

In our previous work, we formalized software copyright licenses and the relations among them by identifying the actions involved in the rights granted by each license and the obligations imposed in return for each right. We placed these actions in a description logic ontology that specified which actions were equivalent, which overlapped, and which were subsumed by which others (in a particular interpretation of the licenses). We analyzed each license into specific rights and their entailed obligations, each involving an actor, a modality (*may*, *need-not*, *must*, *must-not*), an action, and the parameters of the action. With this basis, we annotated each component of an architecture with its license; the component (and in some cases nearby components, other revisions of the component, etc.; see our previous work for a more detailed discussion (Alspaugh et al., 2009, 2011; Alspaugh, Scacchi, & Kawai, 2012) was bound to parameters of the license, and the specific rights and obligations involving that component could then be calculated. From these, it became possible to calculate conflicts and gaps between the rights and obligations for individual components in the architecture, and to calculate the rights (if any) and obligations for the system as a whole. Where there were questions about why a particular right, obligation, conflict, or gap was present, the calculations also provided explanations and (for conflicts or gaps) guidance on possible resolutions.

## Security Licenses

A security license is analogous to an ordinary software copyright license such as GPL. Software licenses consist of intellectual property (IP) rights granted by the licensor, in exchange for corresponding license obligations imposed on the licensee. A license presents the rights that are offered, and for each right enumerates the obligations that are required in order for that right to be granted. Many of the actions required for the obligations are related to the actions allowed by the rights. This is particularly so for open-source licenses, for which fulfilling some of the obligations requires parts of the rights that are granted. Also in open-source licenses, the obligations and rights are framed to take effect in an architectural context, with most obligations taking effect with respect to either the component for which rights are granted or component(s) determined by the connectors and architectural topology around that component. Because software licenses are commonly expressed informally in natural language, the rights and obligations are often presented in an intermingled manner, and much of a license may be devoted to defining terms, classes of entities referred to, and conditions under which the various provisions take effect. But the conceptual structure remains that of a list of rights offered, each in exchange for specific obligations to fulfill.

Our innovation is to similarly but formally specify components' security rights and obligations in restricted natural language, which we can then model, analyze, and support throughout the system's development and evolution, and use to guide its design and instantiation.

Structuring the security policies as licenses gives a form that is more readily accessible to human readers and helps convey intention and rationale by relating each obligation to the right it contributes toward. But we do so in ways that allow for automatic calculation of the interaction of rights and obligations throughout the interconnection neighborhood of each component, the subsystem containing the component, and so on recursively on up to the system as a whole (Alspaugh et al., 2009, 2011). Where the security licenses assigned to the components in the architecture conflict or misalign, automated support can identify the provisions in conflict, locate the conflict to the modules involved, and provide explanations showing the architectural chain of effects that led up to the conflict (Alspaugh et al., 2011). Perhaps most importantly, it supports automation of the analysis of interactions between security measures and of the assessment of the system's overall degree and kind of security as a function of the measures taken for each component, group of components, subsystem, and so forth recursively up to the system as a whole.

We present these possible security rights and obligations as an indication of what sorts of actions might be regulated by security licenses for data organized into security compartments and code organized into components.

### Some Possible Security Rights

1. The right to read/add/remove data in compartment T.
2. The right to replace component C with another component D.
3. The right to update component C to newer version C′.
4. The right to revert component C to older version C′.
5. The right to add/update component C in a specified architectural configuration.
6. The right to alter the architectural topology of subcomponent B.
7. The right to alter the architecture of system S.
8. The right to add/update/remove security mechanism M in a specified configuration.
9. The right to delegate security right R.
10. The right to read the security license of component C.
11. The right to replace the security license L of component C with another security license L′.
12. The right to update security license L.

### Some Possible Security Obligations

1. The obligation for user U to verify his/her identity, by password or other specified authentication process.
2. The obligation for user U to have been vetted by authority A to exercise security right R.
3. The obligation for user U to be delegated a one-time right by authority A to exercise security right R.

4. The obligation for component C to utilize security mechanism M.
5. The obligation for component C to have been vetted by authority A to exercise security right R.
6. The obligation for component C to have been vetted by authority A to be the object of security right R.
7. The obligation for each component connected to component C to utilize security mechanism M.
8. The obligation for each component connected to component C to allow it to exercise security right R.
9. The obligation for security license L to meet specified criteria.
10. The obligation for security license L to be approved by authority A.

## Recent Events

Coordinated international attacks on vulnerable software-intensive systems of high value and controlling complex systems are becoming ever more apparent. As the Stuxnet case demonstrates, security threats to software systems are multi-valent, multi-modal, and distributed across independently developed software system components (Falliere, O Murchu, & Chien, 2011). The Stuxnet attacks entered through software system interfaces at either the component, application subsystem, or base operating system level, and their goal was to go outside or beneath their entry context. However, all of the Stuxnet attacks on the targeted software system could be blocked or prevented through security capabilities associated with OA system interfaces that would (a) limit access or evolutionary update rights lacking proper authorization, as well as (b) “sandboxing” (i.e., isolating) and holding up any evolutionary updates (the attacks) prior to their installation and run-time deployment. Furthermore, as the Stuxnet attack involved the use of corrupted certificates of trust from approved authorities as false credentials that allowed evolutionary system updates to go forward, it seems clear that additional preventions are needed that are orthogonal, external, and prior to, their installation and run-time deployment.

## Discussion and Conclusions

Our efforts outline a number of contributions to this line of research and practice.

First, software security licenses are based on the form and substance of formalized software copyright licenses. As our prior work has formulated and demonstrated, it is possible and practical to systematically develop human readable but computer processable formal specifications of copyright licenses. Subsequently, this paper points the way towards the creation of human readable, yet computationally processable, formal specification of software security policies that can scale to complex system architectures that incorporate components subject to different, heterogenous license rights and obligations.

Second, the ability to create formalized security licenses gives rise to the opportunity to create reusable security licenses that may correspond to common types of security goals or contexts. Here we reference the reusable license framework developed for original IP that are subject to one of the 12 possible copyright licenses available within the Creative Commons framework (Creative Commons, 2012). Such licenses are human readable and allow for automated compliance checking (within limits).

Third, security licenses, like software copyright licenses, can be subject to both legal reasoning and software engineering rationalization. Copyright licenses are enforceable contractual agreements that courts recognize as defining legally-binding relationships between the licensor and licensee. Both compliance with and infringement of a license have economic consequences for licensees and design, configuration, and runtime consequences for developers, integrators, and users. So licenses represent a special class of software engineering construct that informs and constrains the choices made by software developers and integrators. Other areas subject to legal interpretation and engineering rationalization that might benefit from formalizable yet human readable licenses include end-user privacy agreements (regulating what kinds of data may be collected automatically, or by user agreement).

Fourth, if security policies can be expressed in a formalizable, restricted natural language as we indicate, it is possible to develop automated tools and process techniques for specifying, modeling, and analyzing the overall security and integrity of a well-formed software system architecture. In our previous work, we demonstrated tools and techniques that can statically analyze whether a configuration of heterogeneously licensed components matches (i.e., is well-formed), conflicts, or misaligns. A similar form of analysis for OA software can readily determine whether a overall system fails to satisfy its security license constraints, since if so, then the system is insecure and out of compliance by definition. Of course, security licenses do not solve the problem of system security in general, but they do provide a higher-level baseline or standard of excellence than commonly achievable with an informal natural language security policy document.

Fifth, further advances in the creation of software development and run-time environments that continuously assess security license compliance automatically are possible. Such environment can potentially, for example, automate the instantiation software build scripts that comply/enforce license obligations, while enabling security license rights code (e.g., low-level access control capabilities).

Last, software security licenses are an important new class of software engineering construct. Calculations on the formalized security licenses of components in a system architecture can organize and guide the process of designing, developing, and configuring appropriately secure systems. This is a new way software engineering can contribute to developing and sustaining the security of complex, evolving software systems.

## Acknowledgments

This research is supported by grants #N00244-12-1-0004 and #N00244-12-1-0067 from the Acquisition Research Program at the Naval Postgraduate School, and by grant #0808783 from the U.S. National Science Foundation. No review, approval, or endorsement is implied.

## References

Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2009). Intellectual property rights requirements for heterogeneously-licensed systems. In *Proceedings of the 17th IEEE International Requirements Engineering Conference (RE '09)* (pp. 24–33).

Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2011). Presenting software license conflicts through argumentation. In *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011)* (pp. 509–514).

Alspaugh, T. A., Asuncion, H. U., & Scacchi, W. (2012). Software licenses, open source components, and open architectures. In I. Mistrík, A. Tang, R. Bahsoon, & J. A. Stafford (Eds.), *Aligning enterprise, system, and software architectures*. IGI Global.

Alspaugh, T. A., Scacchi, W., & Asuncion, H. U. (2010). Software licenses in context: The challenge of heterogeneously-licensed systems. *Journal of the Association for Information Systems, 11*(11), 730–755.

Alspaugh, T. A., Scacchi, W., & Kawai, R. I. (2012, September). Software licenses, coverage, and subsumption. In *Proceedings of the Fifth International Workshop on Requirements Engineering and Law (RELAW '12)* (pp. 17–24).

Creative Commons. (2012). Retrieved from http://creativecommons.org/

Damianou, N., Dulay, N., Lupu, E., & Sloman, M. (2001). The Ponder policy specification language. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks* (pp. 18–38).

Defense Information Systems Agency. (2011). *Android 2.2 (Dell) security technical implementation guide* (STIG) [Computer software manual].

Falliere, N., O Murchu, L., & Chien, E. (2011). *W32.Stuxnet dossier* (Technical report). Symantec. Retrieved from http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

Franz, M. (2010). E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms (NSPW '10)* (pp. 7–16).

Free Software Foundation. (2007). *GNU General Public License, version 3*. Retrieved from http://www.gnu.org/licenses/gpl-3.0.html

ISO/IEC. (2005). International Standard 27001.

Luo, T., & Du, W. (2011). Contego: Capability-based access control for web browsers. In *Proceedings of the Fourth International Conference on Trust and Trustworthy Computing (TRUST '11)*.

Salamat, B., Jackson, T., Wagner, G., Wimmer, C., & Franz, M. (2011). Runtime defense against code injection attacks using replicated execution. *IEEE Transactions on Dependable and Secure Computing, 8*(4), 588–601.

Scacchi, W., & Alspaugh, T. A. (2012, July). Understanding the role of licenses and evolution in open architecture software ecosystems. *Journal of Systems and Software, 85*(7), 1479–1494.

Scacchi, W., & Alspaugh, T. A. (2013). Advances in the acquisition of secure systems based on open architectures. *Cyber Security and Information Systems Journal, 1*(1). (To appear).

Seacord, R. C. (2008). *CERT C secure coding standard*. Addison-Wesley.

Smalley, S. (2012). *The case for Security Enhanced (SE) Android*. Android Builder's Summit.

Sun, K., Wang, J., Zhang, F., & Stavrou, A. (2012). SecureSwitch: BIOS-assisted isolation and switch between trusted and untrusted commodity OSes. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS 2012)*.

Uszok, A., Bradshaw, J. M., Johnson, M., et al. (2004). KAoS policy management for semantic web services. *IEEE Intelligent Systems, 19*(4), 32–41.

Xen hypervisor project. (2012). Retrieved from http://xen.org/products/xenhyp.html