Revised version to appear in *Proc. 8th IFIP International Conference on Open Source Systems*, Hamammet, Tunisia, September 2012.

Designing Secure Systems Based on Open Architectures with Open Source and Closed Source Components

Walt Scacchi and Thomas A. Alspaugh
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
wscacchi@ics.uci.edu, thomas.alspaugh@acm.org
http://www.ics.uci.edu/~wscacchi

Abstract. The development and evolution of secure open architecture systems has received insufficient consideration. Such systems are composed of both open source and closed software software components subject to different security requirements in an architecture in which evolution can occur by evolving existing components, replacing them, or refactoring their interfaces, interconnections and configuration. But this may result in possible security requirements conflicts and organizational liability for failure to fulfill security obligations. We are developing an approach for understanding and modeling software security requirements as "security licenses", as well as for analyzing conflicts among groups of such licenses in realistic system contexts and for guiding the acquisition, integration, or development of systems with open source components in such an environment. Consequently, this paper reports on our efforts to extend our existing approach to specifying and analyzing software Intellectual Property (IP) licenses to now address software security licenses that can be associated with secure OA systems.

1. Introduction

A growing number of enterprises are adopting a strategy in which a software-intensive system is developed with an open architecture (OA) [20,2,4,22], whose components may be open source software (OSS) or closed source 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 copyright license. With this approach, the system development organization becomes an integrator of components largely produced elsewhere that are interconnected through middleware or open APIs as necessary to achieve the desired result.

An OA development process arises in a software ecosystem in which the integrator is influenced from one direction by the goals, interfaces, license choices, and release cycles of the component producers, and in another direction by the needs of its consumers. As a result the software components are reused more widely, and the

resulting OA systems can achieve reuse benefits such as reduced costs, increased reliability, and potentially increased agility in evolving to meet changing needs. An emerging challenge is to realize the benefits of this approach when the individual components are subject to different security requirements.

We have been able to address an analogous problem of how to specify and analyze the Intellectual Property (IP) rights and obligations of the licenses of software components [2,3,4,5]. Our efforts now focus on the challenge of how to specify and analyze software components and composed system security rights and obligations using a new information structure we call a "security license." Alternative renderings for a security license are beyond the scope of this paper, but at this point, we believe it is appropriate to develop candidate security policy expressions that can be incorporated into security licenses. Further, we seek to articulate security license terms and conditions in ways that can be easily formalized, readily applied to large-scale OA systems, as well as be automatically analyzed or tested in ways we have already demonstrated [4,5]. This is another goal of our research here.

Next, the challenge of specifying secure software systems composed from secure or insecure components is inevitably entwined with the software ecosystems that arise for OA systems. An example software ecosystem producing and integrating software components subject to different security practices is portrayed in Figure 1. We find that an OA software ecosystem involves not only organizations and individuals producing and consuming components, and supply paths from producer to consumer; but also:

- the OA of the system(s) in question, and how best to secure it,
- the open interfaces provided by the components, and how to specify component security requirements that are enforceable or satisfiable at the interface level,
- the evolution of related components that can be assessed in terms of how overall system security rights and obligations may change, and
- the rights and obligations resulting from the security licenses under which various components are released, that propagate from producers to consumers.

In order to most effectively use an OA approach in developing and evolving a system, it is essential to consider its OA ecosystem. An OA system draws on components from proprietary closed source software vendors and open source software projects. Its architecture is bounded and facilitated by the relevant ecosystem of producers, from which the initial components are chosen. The choice of a specific OA begins a specialized software ecosystem involving components that meet (or can be encapsulated or "wrapped" to meet) the open interfaces used in the architecture. We do not claim this is the best or the only way to reuse components or produce secure OA systems, but it is an ever more widespread way. In this paper we build on previous work on heterogeneously-licensed systems [15, 22, 2] by

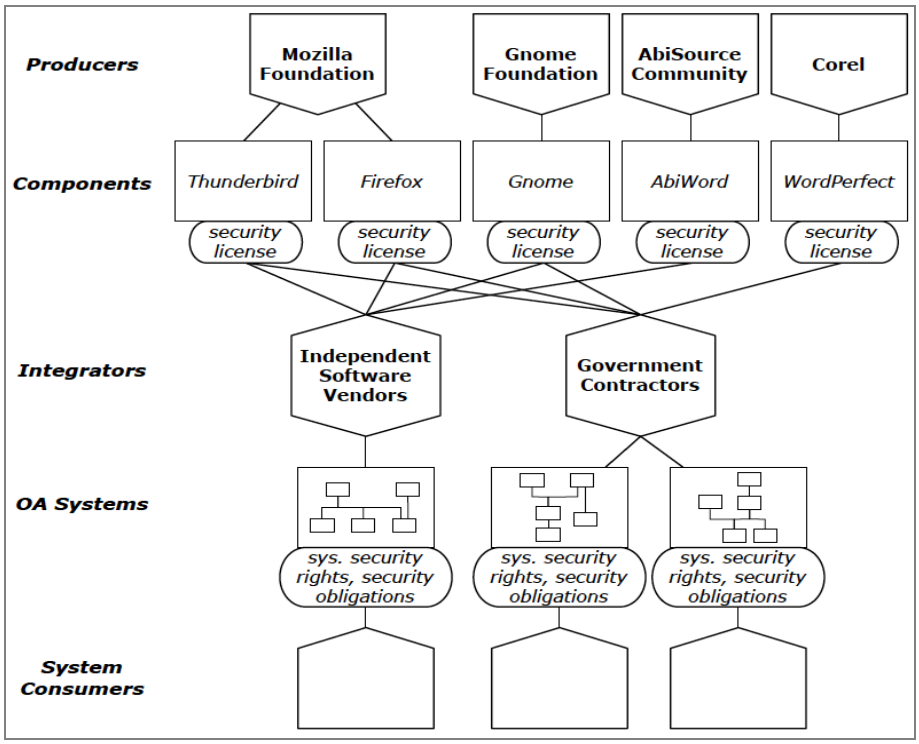

Figure 1: A sample software ecosystem in which secure OA systems may be developed.

examining the role of security licenses for components included within OA software ecosystems.

In the remainder of this paper, we survey some related work (Section 2), define and examine characteristics of open architectures with or without secure software elements (Section 3), define and examine characteristics for how secure OA systems evolve (Section 4), introduce a structure for security licenses (Section 5), outline security license architectures (Section 6), and sketch our approach for security license analysis (Section 7). We then close with our conclusions (Section 8).

2. Related Work

Software systems, whether operating as standalone components, applications, or as elements within large system compositions, are continuously being subjected to security attacks. These attacks seek to slip through software vulnerabilities known to the attackers but perhaps not by the software component producers, system integrators or consumers. These attacks often seek to access, manipulate, or remotely affect the data values or control signals that a component or composed system

processes for nefarious purposes, or seek to congest or over-saturate networked services. Recent high profile security attacks like *Stuxnet* [11] reveal that security attacks may be very well planned and employ a bundle of attack vectors and social engineering tactics in order for the attack to reach strategic systems that are mostly isolated and walled off from public computer networks. The Stuxnet attack entered through software system interfaces at either the component, application subsystem, or base operating system level (e.g., via removable thumb drive storage devices), and their goal was to go outside or beneath their entry context. Furthermore, as the Stuxnet attack involved the use of corrupted certificates of trust from approved authorities as false credentials that allowed corrupt evolutionary system updates to go forward, it seems clear that additional preventions are needed that are external to, and prior to, their installation and run-time deployment. In our case, that means we need to specify and analyze software security requirements and evolutionary update capabilities at architectural *design-time* and system integration *build-time*, and then reconcile those with the *run-time* system composition. It also calls for the need to maintain the design-time, build-time, and run-time system compositions in repositories remote from system installations, and then cross-checked and independently verified prior to run-time deployment in a high security system application.

As already noted, both software intellectual property licenses and security 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 systems or system components as intellectual property (IP) or security requirements (i.e., capabilities) 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 [8, 9]. Traditionally, developing robust specifications for non-functional software system security properties in natural language often produces specifications that are ambiguous, misleading, inconsistent across system components, and lacking sufficient details [23]. Using a semantic model to formally specify the rights and obligations required for a software system or component to be secure [8, 9, 23] 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 development — design-time, build-time, and run-time. The approach we have been developing for the past few years for modeling and analyzing software system IP license architectures for OA systems [3, 4, 5, 22], may therefore be extendable to also being able to address OA systems with

heterogeneous “software security license” rights and obligations. Furthermore, the idea of common or reusable software security licenses may be analogous to the reusable security requirements templates proposed by Firesmith [13]. But such an extension of the semantic software license modeling, meta-modeling, and computational analysis tools to also support software system security can be recognized as a next stage of our research studies.

3. Secure Open Architecture Composition

Open architecture (OA) software is a customization technique introduced by Oreizy [20] and further expanded [2,4,5,22] that enables third parties to modify a software system through its explicitly modeled architecture, evolving the system by replacing its components. Increasingly more software-intensive systems are developed using an OA strategy, not only with open source software (OSS) components but also proprietary components with open APIs. Similarly, these components may or not have their own security requirements that must be satisfied during their build-time integration or run-time deployment, such as registering the software component for automatic update and installation of new software versions that patch recently discovered security vulnerabilities or prevent invocation of known exploits. Using this approach can lower development costs and increase reliability and function, as well as adaptively evolve software security [22]. Composing a system with heterogeneously secured components, however, increases the likelihood of conflicts, liabilities, and no-rights stemming from incompatible security requirements. Thus, in our work we define a secure OA system as *a software system consisting of components that are either open source or proprietary with open API, whose overall system rights at a minimum allow its use and redistribution, in full or in part such that they do not introduce new security vulnerabilities at the system architectural level*.

It may appear that using a system architecture that incorporate secure OSS and proprietary components, and uses open APIs, will result in a secure OA system. But not all such architectures will produce a secure OA, since the (possibly empty) set of available license rights for an OA system depends on: (a) how and why secure or insecure components and open APIs are located within the system architecture, (b) how components and open APIs are implemented, embedded, or interconnected, and (c) the degree to which the IP and security licenses of different OSS components encumber all or part of a software system’s architecture into which they are integrated [22, 1].

The following kinds of software elements appearing in common software architectures can affect whether the resulting overall composed systems are open or closed, as well as compliant with specified security policies (rights and obligations propagated from components to the overall system) [6].

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, (d) intra-application script code, as for creating Rich Internet Applications using domain-specific languages such as XUL for the Firefox Web browser [12] or "mashups" [19]. whose source code is available and they can be rebuilt, or (e) similar script code that can either install and invoke externally developed plug-in software components, or invoke external application (helper) components. Each may have its own distinct IP/security requirements.

Executable components—These components are in binary form, and the source code may not be open for access, review, modification, or possible redistribution [21]. 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 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" [18].

Software connectors—Software whose intended purpose is to provide a standard or reusable way of communication through common interfaces, e.g. High Level Architecture [17], CORBA, MS .NET, Enterprise Java Beans, and GNU Lesser General Public License (LGPL) libraries. Connectors can also limit the propagation of IP license obligations, mandate the propagation of license obligations (e.g, via use of a licenses like the Affero GPL), or provide additional security capabilities.

Methods of connection—These include linking as part of a configured subsystem, dynamic linking, and client-server connections. 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, and whose internal architecture may comprise components with different licenses, affecting the overall system license and its security requirements. 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 shims, or other connectors that block the propagation of reciprocal obligations.

Figure 2 shows a high-level run-time view of a composed OA system whose reference architectural design in Figure 3 includes all the kinds of software elements listed above. This reference architecture has been instantiated in a build-time configuration in Figure 4 that in turn could be realized in alternative run-time configurations in Figures 5, 6, 7 with different security capabilities (policies) and overall system security schemes.The configured systems consist of software components such as a Mozilla Web browser, Gnome Evolution email client, and AbiWord word processor (similar to MS Word), all running on a RedHat Fedora Linux operating system accessing file, print, and other remote networked servers

such as an Apache Web server. Components are interconnected through a set of software connectors that bridge the interfaces of components and combine the provided functionality into the system's services. However, note how the run-time software architecture does not pre-determine how security capabilities will be assigned and distributed across different variants of the run-time composition.

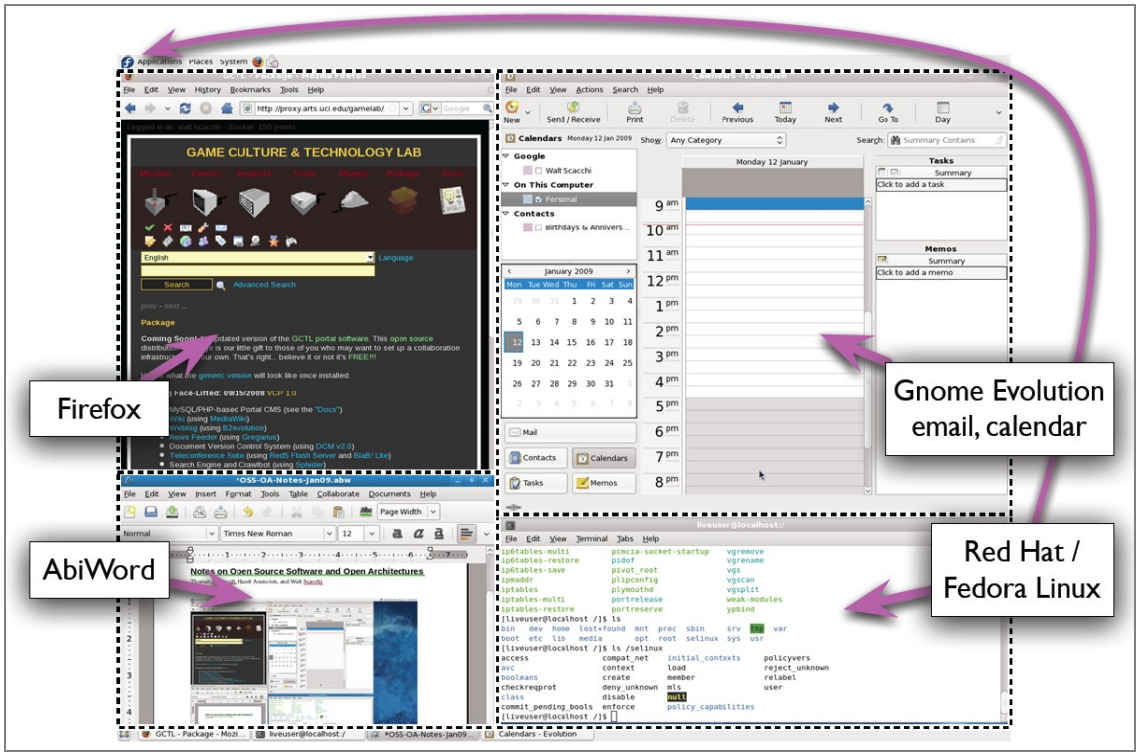

Figure 2: An example composite OA system potentially subject to different IP and security licenses.

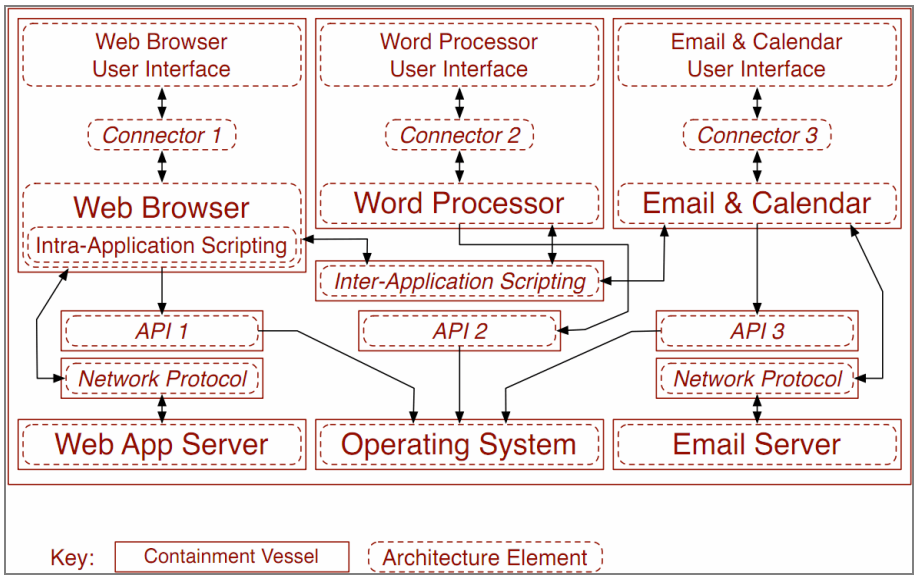

Figure 3: The design-time architecture of the system in Figure 2 that specifies a required security containment vessel (domain) scheme.

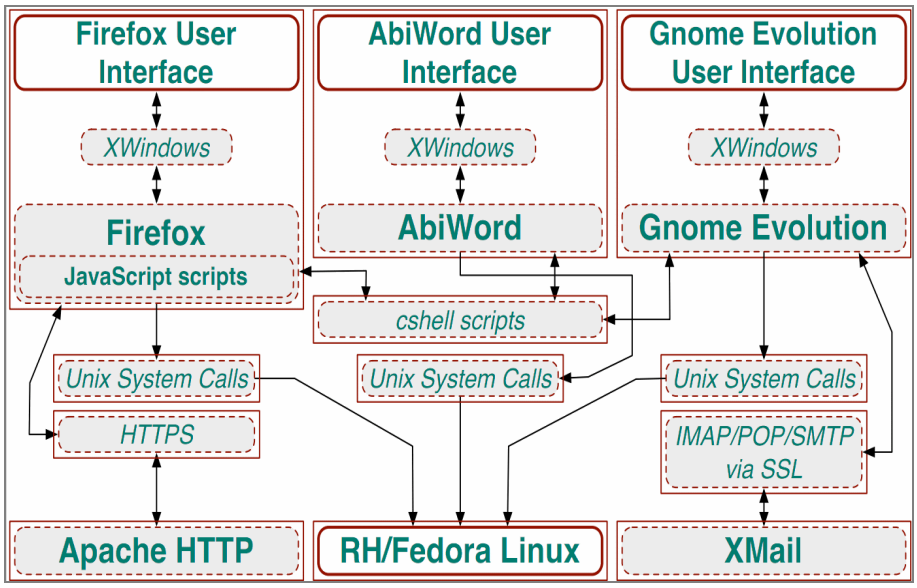

Figure 4: A secure build-time architecture describing the version running in Figure 2 with a specified security containment vessel scheme.

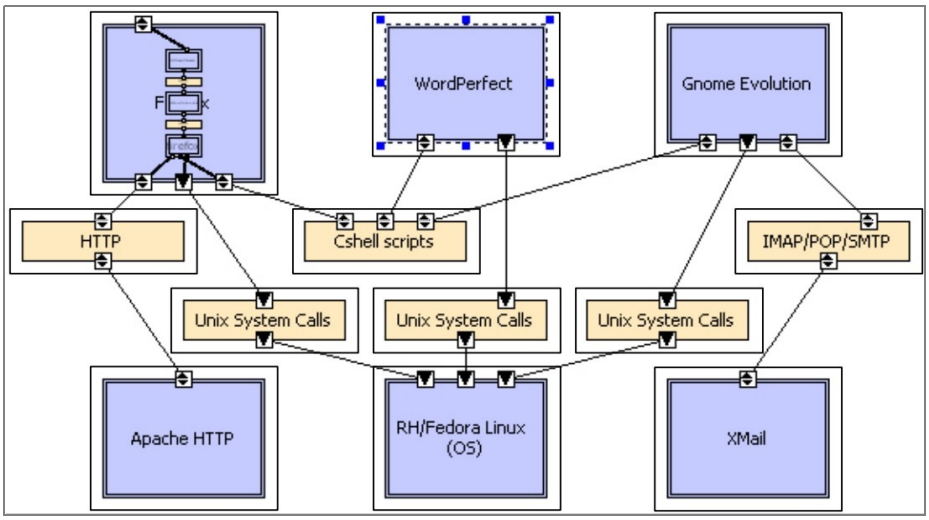

Figure 5: Instantiated build-time OA system with maximum security architecture of Figure 4 via individual security containment vessels (domains) for each system element.

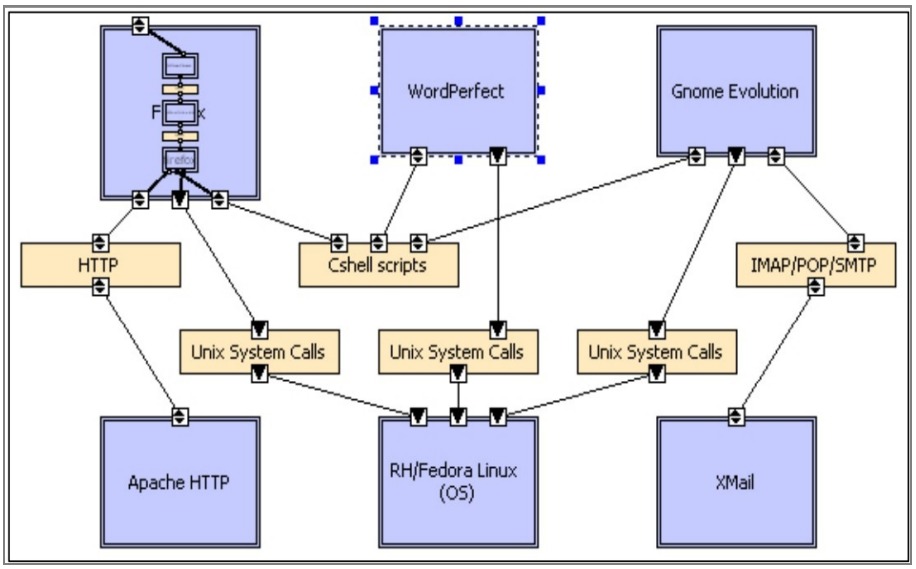

Figure 6: Instantiated build-time OA system with minimum security architecture of Figure 4 via a single overall security containment vessel (domain) for the complete system using a common software hypervisor.

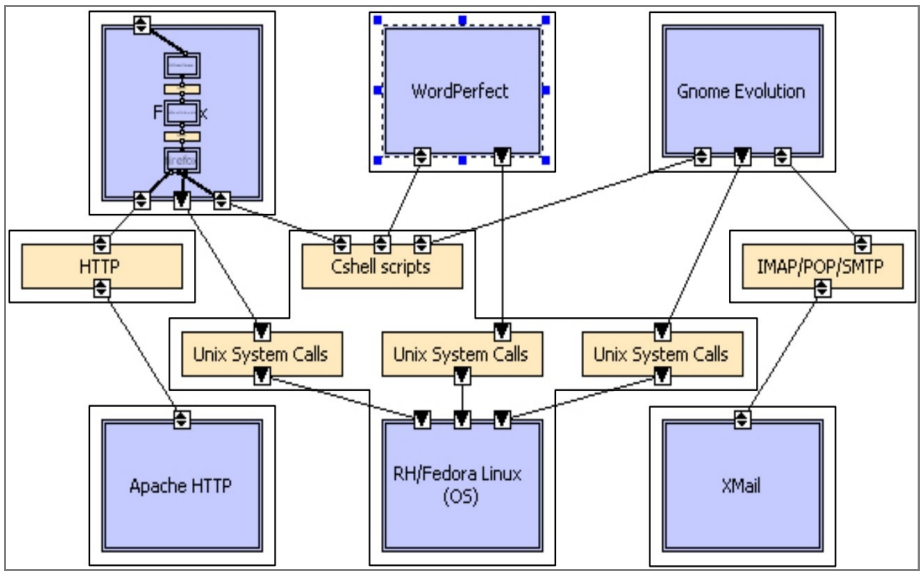

Figure 7: Instantiated build-time OA system with mixed security architecture of Figure 4 via security containment vessels (domains) for some groupings of system elements.

4. OA System Evolution

An OA system can evolve by a number of distinct mechanisms, some of which are common to all systems but others of which are a result of heterogeneous IP and security licenses in a single system.

By component evolution— One or more components can evolve, altering the overall system's characteristics (for example, upgrading and replacing the *Firefox* Web browser from version 3.5 to 3.6 which may update existing software functionality while also patching recent security vulnerabilities).

By component replacement— One or more components may be replaced by others with different behaviors but the same interface, or with a different interface and the addition of shim code to make it match (for example, replacing the GPL'd *AbiWord* word processor with either *Open Office* or *MS Word*, perhaps depending on which is considered the least vulnerable to security attack).

By architecture evolution— The OA can evolve, using the same components but in a different configuration, altering the system's characteristics. For example, as discussed in Section 3, changing the configuration in which a component is connected can change how its IP or security license affects the rights and obligations for the overall system. This could arise when replacing email and word processing applications with web services like *Google Mail* and *Google Docs*, which we might

assume may be more secure since the Google services (operating in a cloud environment) may not be as easily accessed or penetrated by a security attack.

By component license evolution— The license under which a component is available may change, as for example when the license for the Mozilla core components was changed from the Mozilla Public License (MPL) to the current Mozilla Disjunctive Tri-License; or the component may be made available under a new version of the same license, as for example when the GNU General Public License (GPL) version 3 was released. Similarly, the security license for a component may be changed by its producers, or the security license for a composed system changed by its integrators, in order to prevent or deter recently discovered security vulnerabilities or exploits before an evolutionary version update (or patch) can be made available.

By a change to the desired rights or acceptable obligations— The OA system's integrator or consumers may desire additional IP or security 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, whether by changing components, evolving the architecture, or other means, to provide the desired rights within the scope of the acceptable obligations. For example, they may no longer be willing or able to provide the source code for components that have known vulnerabilities that have not been patched and eliminated.

The interdependence of integrators and producers results in a co-evolution of software within an OA ecosystem. Closely-coupled components from different producers must evolve in parallel in order for each to provide its services, as evolution in one will typically require a matching evolution in the other. Producers may manage their evolution with a loose coordination among releases, for example as between the Gnome and Mozilla organizations. Each release of a producer component create a tension through the ecosystem relationships with consumers and their releases of OA systems using those components, as integrators accommodate the choices of available, supported components with their own goals and needs. As discussed in our previous work [2], license rights and obligations are manifested at each component's interface, then mediated through the system's OA to entail the rights and corresponding obligations for the system as a whole. As a result, integrators must frequently re-evaluate an OA system's IP/security rights and obligations. In contrast to homogeneously-licensed systems, license change across versions is a characteristic of OA ecosystems, and architects of OA systems require tool support for managing the ongoing licensing changes [3,4,5].

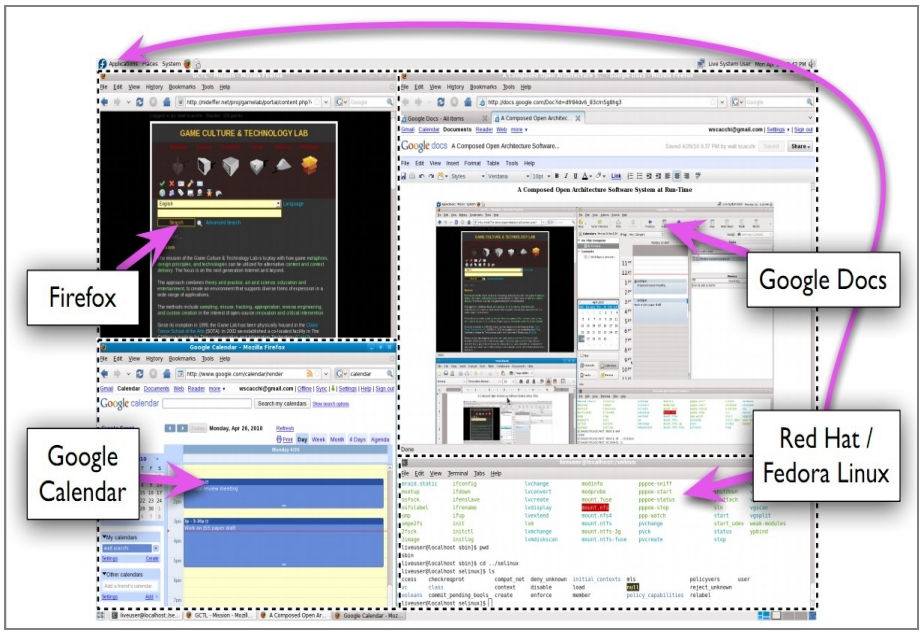

Figure 8: A second instantiation at run-time (Firefox, Google Docs and Calendar operating within different Firefox run-time sessions, Fedora) of the OA system in Figures 2, 3, and 4 as an evolutionary alternative system version, which in turn implies or requires an alternative security containment scheme.

We propose that such support must have several characteristics.

- It must rest on a license structure of rights and obligations (Section 5), focusing on obligations that are enactable and testable.
- It must take account of the distinctions between the design-time, build-time, and distribution-time architectures (Sections 3, 5, 6) and the rights and obligations that come into play for each of them.
- It must distinguish the architectural constructs significant for software licenses, and embody their effects on rights and obligations (Section 3).
- It must define license architectures (Section 6).
- It must provide an automated environment for creating and managing license architectures. We are developing a prototype that manages a license architecture as a view of its system architecture [2,4,5].
- Finally, it must automate calculations on system rights and obligations so that they may be done easily and frequently, whenever any of the factors affecting rights and obligations may have changed (Section 7).

5. Security Licenses

Licenses typically impose obligations that must be met in order for the licensee to realize the assigned rights. Common IP/copyright license obligations include the obligation to publish at no cost any source code you modify (MPL) or the reciprocal obligation to publish all source code included at build-time or statically linked (GPL). The obligations may conflict, as when a GPL'd component's reciprocal obligation to publish source code of other components is combined with a proprietary component's license prohibition of publishing its source code. In this case, no rights may be available for the system as a whole, not even the right of use, because the two obligations cannot simultaneously be met and thus neither component can be used as part of the system. Security capabilities can similarly be expressed and bound to the data values and control signals that are visible in component interfaces, or through component connectors.

Some typical security rights and obligations might be:

- The right to read data in containment vessel T.
- The right to replace specified component C with some other component.
- The right to add or update specified component D in a specified configuration.
- The right to add, update, or remove a security mechanism.
- The obligation for a specific component to have been vetted for the capability to read and update data in containment vessel T.
- The obligation for a user to verify his/her authority to see containment vessel T, by password or other specified authentication process.

The basic relationship between software IP/security license rights and obligations can be summarized as follows: if the specified obligations are met, then the corresponding rights are granted. For example, if you publish your modified source code and sub-licensed derived works under MPL, then you get all the MPL rights for both the original and the modified code. Similarly, software security requirements are specified as security obligations that when met, allow designated users or other software programs to access, modify, and redistribute data and control information to designated repositories or remote services. However, license details are complex, subtle, and difficult to comprehend and track—it is easy to become confused or make mistakes. The challenge is multiplied when dealing with configured system architectures that compose a large number of components with heterogeneous IP/security licenses, so that need for legal counsel begins to seem inevitable [21, 14].

We have developed an approach for expressing software licenses of different types (intellectual property and security requirements) that is more formal and less ambiguous than natural language, and that allows us to calculate and identify conflicts arising from the rights and obligations of two or more component's licenses. Our approach is based on Hohfeld's classic group of eight fundamental

jural relations [16], of which we use right, duty, no-right, and privilege. We start with a tuple <*actor, operation, action, object*> for expressing a right or obligation. The actor is the "licensee" for all the licenses we have examined. The operation is one of the following: "may", "must", "must not", or "need not", with "may" and "need not" expressing rights and "must" and "must not" expressing obligations. The action is a verb or verb phrase describing what may, must, must not, or need not be done, with the object completing the description. A license may be expressed as a set of rights, with each right associated with zero or more obligations that must be fulfilled in order to enjoy that right. Figure 9 shows the meta-model with which we express licenses.

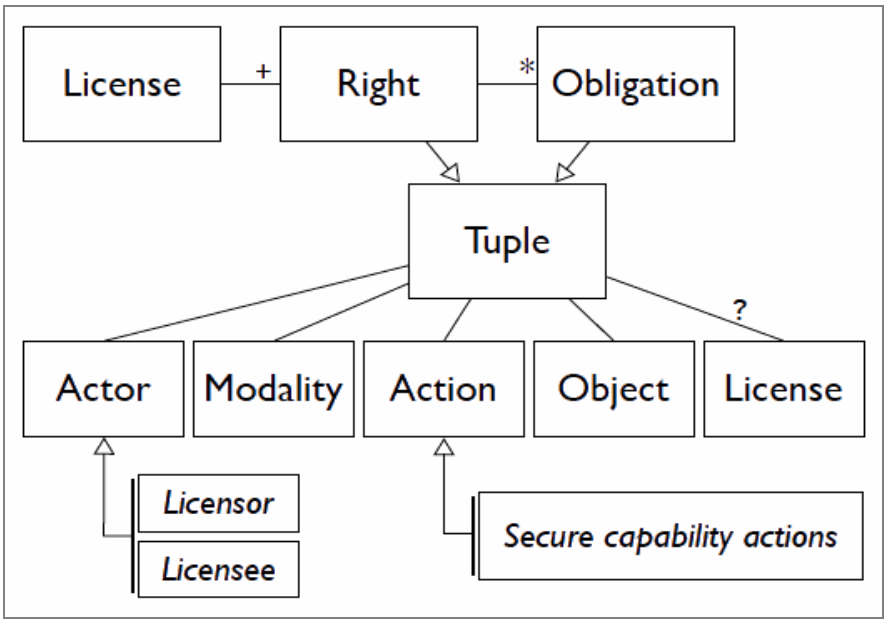

Figure 9: Security license meta-model

Designers of secure systems have developed a number heuristics to guide architectural design in order to satisfy overall system security requirements, while avoiding conflicts among interacting security mechanisms or defenses. However, even using design heuristics (and there are many), keeping track of security rights and obligations across components that are interconnected in complex OAs quickly becomes too cumbersome. Automated support is needed to manage the complexity of multi-component system compositions where different security requirements must be addressed through different security capabilities.

6. Security License Architectures

Our security license model forms a basis for effective reasoning about licenses in the context of actual systems, and calculating the resulting rights and obligations. In order to do so, we need a certain amount of information about the system's configuration at design-time, build-time, and run-time deployment. The needed information comprises the license architecture, an abstraction of the system architecture:

1. the set of components of the system (for example, see Figure 2) for the current system configuration, as well as subsequently for system evolution update versions (as seen in Figure 8);
2. the relation mapping each component to its security requirements (specified and analyzed at design-time, as exemplified in Figure 3) or capabilities (specified and analyzed at build-time in Figure 4 and run-time across alternatives shown in Figure 5, 6, and 7);
3. the connections between components and the security requirements or capabilities of each connector passing data or control signals to/from it; and
4. possibly other information, such as information to detect or prevent IP and security requirements conflicts, which is as yet undetermined.

With this information and definitions of the licenses involved, it should possible to automatically calculate rights and obligations for individual components or for the entire system, as well as guide/assess system design and evolution, using an automated environment of the kind that we have previously demonstrated [2, 3, 4, 5].

7. Security License Analysis

Given a specification of a software system's architecture, we can associate security license attributes with the system's components, connectors, and sub-system architectures, resulting in a license architecture for the system, and calculate the security rights and obligations for the system's configuration. Due to the complexity of license architecture analysis, and the need to re-analyze every time a component evolves, a component's security license changes, a component is substituted, or the system architecture changes, OA integrators really need an automated license architecture analysis environment. We have developed a prototype of such an environment for analogous calculations for software copyright licenses [3, 5], and are extending this approach to analyze security licenses. But here we identify two types of analysis that are representative of those our approach supports.

7.1 Security obligation conflicts

A security obligation can conflict with another obligation, a related right for the same or nearby components, or with the set of available security rights, by requiring a

right that has not been granted. For instance, consider two connected components C and D with security obligations:

(**O1**) *The obligation for component C to have been vetted for the capability to read and update data in containment vessel T.*

(O2) The obligation for all components connected to specified component D to grant it the capability to read and update data in containment vessel T.

If C has not been vetted, then these two obligations conflict. This possible conflict must be taken into consideration in different ways at different development times:

- at design time, ensuring that it will be possible to vet C;
- at build time, ensuring that the specific implementation of C has been vetted successfully; and
- possibly at run time as well, confirming that C is certified to have been vetted, or (if C is dynamically connected at run time) vetting C before trusting this connection to it.

The second obligation may also conflict with the set of available security rights, for example if D is connected to component E for which the security right:

(**R1**) *The right to read and update data in containment vessel T using component E is not available.*

The absence of such conflicts does not mean, of course, that the system is secure. But the presence of conflicts reliably indicates that it is not secure.

7.2 Rights and obligations calculations

The rights available for the entire system (the right to read and update data in containment vessel T, the right to replace components with other components, the right to update component security licenses, etc.) are calculated as the intersection of the sets of security rights available for each component of the system. If a conflict is found involving the obligations and rights of interacting components, it is possible for the system architect to consider an alternative scheme, e.g. using one or more connectors along the paths between the components that act as a security firewall. This means that the architecture and the automated environment together can determine what OA design best meets the problem at hand with available software components. Components with conflicting security licenses do not need to be arbitrarily excluded, but instead may expand the range of possible architectural alternatives if the architect seeks such flexibility and choice.

8. Conclusion

This paper introduces the concept and initial scheme for systematically specifying and analyzing the security requirements for complex open architecture systems. We argue that such requirements should be expressed as operational capabilities that can

be collected and sequenced within a new information structure we call a security license. Such a license expresses security in terms of capabilities that provide users or programs obligations and rights for how they may access data or control information, as well as how the may update or evolve system elements. These security license rights and obligations thus play a key role in how and why an OA system evolves in its ecosystem of software component producers, system integrators and consumers.

We note that changes to the license obligations and rights, whether for control of intellectual property or software security, across versions of components is a characteristic of OA systems whose components are subject to different security requirements or other license restrictions. A structure for modeling software licenses and automated support for calculating its rights and obligations are needed in order to manage an OA system's evolution in the context of its ecosystem.

We have outlined an approach for achieving these and sketched how they further the goal of reusing components in developing software-intensive systems. Much more work remains to be done, but we believe this approach turns a vexing problem into one for which workable, as well as robust formal, solutions can be obtained.

10 . Acknowledgments

This research is supported by grants #N00244-10-1-077 and #N00244-12-1-0004 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 implied.

References

[1] T. A. Alspaugh and A. I. Anton. Scenario support for effective requirements. *Information and Software Technology*, 50(3):198–220, Feb. 2008.

[2] T. A. Alspaugh, H. U. Asuncion, and W. Scacchi. Analyzing software licenses in open architecture software systems. In *2nd International Workshop on Emerging Trends in FLOSS Research and Development* (FLOSS), May 2009.

[3] T. A. Alspaugh, H. U. Asuncion, and W. Scacchi. Intellectual property rights requirements for heterogeneously-licensed systems. In *17th IEEE International Requirements Engineering Conference* (RE'09), pages 24–33, Aug. 31–Sept. 4 2009.

[4] T. A. Alspaugh, H. U. Asuncion, and W. Scacchi. The Challenge of Heterogeneously Licensed Systems in Open Architecture Software Ecosystems, In S.Jansen, S. Brinkkemper, and M. Cusamano (Eds). *Software Ecosystems*, (to appear, 2012).

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

[6] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[7] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[8] T. D. Breaux and A. I. Anton. Analyzing goal semantics for rights, permissions, and obligations. In RE '05: *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, pages 177–188, 2005.

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

[10] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[11] N. Falliere, L. O Murchu, and E. Chien. *W32.Stuxnet dossier, Version 1.4,* February 2011, http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

[12] K. Feldt. *Programming Firefox: Building Rich Internet Applications with XU*L. O'Reilly Media, Inc., 2007.

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

[14] R. Fontana, B. M. Kuhn, E. Moglen, M. Norwood, D. B. Ravicher, K. Sandler, J. Vasile, and A. Williamson. *A Legal Issues Primer for Open Source and Free Software Projects*. Software Freedom Law Center, 2008. http://www.softwarefreedom.org/resources/2008/foss-primer.html

[15] D. M. German and A. E. Hassan. License integration patterns: Dealing with licenses mis-matches in component-based development. In *28th International Conference on Software Engineering* (ICSE '09), May 2009.

[16] W. N. Hohfeld. Some fundamental legal conceptions as applied in judicial reasoning. *Yale Law Journal*, 23(1):16–59, Nov. 1913.

[17] F. Kuhl, R. Weatherly, and J. Dahmann. *Creating computer simulation systems: an introduction to the high level architecture*. Prentice Hall, 1999.

[18] B. C. Meyers and P. Oberndorf. *Managing Software Acquisition: Open Systems and COTS Products*. Addison-Wesley Professional, 2001.

[19] L. Nelson and E. F. Churchill. Repurposing: Techniques for reuse and integration of interactive systems. In *International Conference on Information Reuse and Integration* (IRI-08), page 490, 2006.

[20] P. Oreizy. *Open Architecture Software: A Flexible Approach to Decentralized Software Evolution*. PhD thesis, University of California, Irvine, 2000.

[21] L. Rosen. *Open Source Licensing: Software Freedom and Intellectual Property Law*. Prentice Hall, 2005.

[22] W. Scacchi and T. A. Alspaugh. Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In *5th Annual Acquisition Research Symposium*, May 2008.

[23] S. S. Yau and Z. Chen. A framework for specifying and managing security requirements in collaborative systems. In *Third International Conference on Autonomic and Trusted Computing* (ATC 2006), pages 500–510, 2006.