UCI-AM-18-029

ACQUISITION RESEARCH PROGRAM
SPONSORED REPORT SERIES

Cybersecure Modular Open Architecture Software Systems for Stimulating Innovation

8 December 2017

Dr. Walt Scacchi

Dr. Thomas A. Alspaugh

Institute for Software Research

University of California, Irvine

Disclaimer: This material is based upon work supported by the Naval Postgraduate School Acquisition Research Program under Grant No. N00244-16-1-0053. The views expressed in written materials or publications, and/or made by speakers, moderators, and presenters, do not necessarily reflect the official policies of the Naval Postgraduate School nor does mention of trade names, commercial practices, or organizations imply endorsement by the U.S. Government.

Approved for public release; distribution is unlimited.

Prepared for the Naval Postgraduate School, Monterey, CA 93943.

The research presented in this report was supported by the Acquisition Research Program of the Graduate School of Business & Public Policy at the Naval Postgraduate School.

To request defense acquisition research, to become a research sponsor, or to print additional copies of reports, please contact any of the staff listed on the Acquisition Research Program website (www.acquisitionresearch.net).

Executive Summary

This research investigated a new approach to stimulate innovation in the acquisition, production and evolution of cybersecure modular OA software systems. These systems increasingly incorporate Web-based, mobile, or low-cost microelectronic devices. Systems of these kinds must combine best-of-breed software components subject to agile, adaptive requirements of multiple parties, while conforming to reusable software products lines. We seek to make this a simpler, more transparent, and more tractable process. Our recent and continuing line of research studies, publications and reports demonstrate how complex OA systems can be designed, built, and deployed with alternative components and connectors resulting in functionally similar system versions, to satisfy overall system capability requirements as well as individual OA system component intellectual property (IP) and cybersecurity requirements. These requirements are surfacing new challenges that can decrease (or increase) software acquisition costs.

Our next step addressed here was to initiate investigations the use of *smart contracts and associated technologies* (e.g., cryptocurrency, domain-specific blockchain transaction languages and computational tools) for specifying shared agreements between multiple parties to acquisition efforts. We believe smart contracts can be computationally enacted during the design, integration, release, deployment, and evolution of cybersecure, modular open architecture software systems in ways that can model, track and analyze the associated contractual obligations and customer rights that drive costs and risks. Smart contracts incorporate computational specifications (i.e., computer programming script code) that enable formal and precise agreements between parties that can entail costing constraints, and production or cybersecurity requirements, that are associated with articulated OA system procurement obligations and rights. The associated technologies for smart contracts are emerging capabilities that enable computational protocols for tracking elemental transactions between multiple parties to a shared contractual agreement. Such agreements can arise, for example, when different commercial firms, non-profit enterprises, program

offices, and government agencies decide to share acquisition costs and risks in order to more rapidly assemble, produce, deliver, or evolve innovative cybersecure modular OA software systems. Our research results are documented in this Final Report.

Last, our research results have been well received in presentations to different audiences, including academic and industry research groups, the larger Defense community, and the Federal Government more broadly. In particular, throughout 2017 our research results have been presented to audiences at the 2017 Acquisition Research Symposium (Monterey, CA). Other project activities that produced material results include multiple presentations at the new Cybersecurity Policy & Research Institute based at the University of California, Irvine. These presentations have included senior level executives from more than 80 industry and local government agencies, including law enforcement programs now burdened with investigating cybercrimes that entail covert entry, data exfiltration, and extortion based on legacy systems. As can been seen in these chapters, common and differentiated research results found in the chapters represent our efforts at reaching out to different audiences interested in our research, and what advice or guidance it may offer to such audiences.

Acknowledgement

This report was supported by grant #N00244-6-1-0053 from the Acquisition Research Program at the Naval Postgraduate School, Monterey, CA. 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 persons, group, enterprise, or government agency.

THIS PAGE INTENTIONALLY LEFT BLANK

About the Authors

Walt Scacchi—is 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 1981–1998, 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 200 research papers, and has directed 70 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. [wscacchi@ics.uci.edu]

Thomas 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. [thomas.alspaugh@acm.org]

THIS PAGE INTENTIONALLY LEFT BLANK

UCI-AM-18-029

ACQUISITION RESEARCH PROGRAM
SPONSORED REPORT SERIES

Cybersecure Modular Open Architecture Software Systems for Stimulating Innovation

8 December 2017

Dr. Walt Scacchi

Dr. Thomas A. Alspaugh

Institute for Software Research

University of California, Irvine

Disclaimer: This material is based upon work supported by the Naval Postgraduate School Acquisition Research Program under Grant No. N00244-16-1-0053. The views expressed in written materials or publications, and/or made by speakers, moderators, and presenters, do not necessarily reflect the official policies of the Naval Postgraduate School nor does mention of trade names, commercial practices, or organizations imply endorsement by the U.S. Government.

THIS PAGE LEFT INTENTIONALLY BLANK

Table of Contents

Research Overview

Introduction

The goal of this research was to investigate a new approach to stimulate innovation in the acquisition, production and evolution of cybersecure modular Open Architecture (OA) software systems [Kendall 2015]. We seek to make this a simpler, more transparent, and more tractable process. Our recent research demonstrates how complex OA systems can be designed, built, and deployed with alternative components and connectors resulting in functionally similar system versions, to satisfy overall system capability requirements as well as individual OA system component intellectual property (IP) and cybersecurity requirements [DoDGSA 2015, Scacchi and Alspaugh 2011,Scacchi and Alspaugh 2012, Scacchi and Alspaugh 2013a, Scacchi and Alspaugh 2013b, 20]. These requirements are surfacing new challenges that can decrease (or increase) software acquisition costs [Scacchi and Alspaugh 2014, Scacchi and Alspaugh 2015, Scacchi and Alspaugh 2016].

Our next step proposed here was to investigate the use of, and technical risks for, *blockchains* [2017], *smart contracts* [2017], and *associated technologies.* The associated technologies include distributed ledgers, cryptocurrency, domain-specific blockchain transaction languages and computational tools [Ethereum 2017]) for specifying shared agreements between multiple parties to acquisition efforts. We believe smart contracts can be computationally enacted during the design, integration, release, deployment, and evolution of cybersecure, modular open architecture software systems [Scacchi and Alspaugh 2013b, Scacchi and Alspaugh 2015, Scacchi and Alspaugh 2016] in ways that can model, track and analyze the associated contractual obligations and customer rights that drive costs and risks. Smart contracts incorporate computational specifications (i.e., computer programming script code) that enable formal and precise agreements between parties that can entail costing constraints, and production or cybersecurity requirements, that are associated with articulated OA system procurement obligations and rights. The associated technologies for smart contracts are emerging capabilities that enable computational protocols for tracking elemental transactions between multiple parties to a shared contractual agreement. Such agreements can arise, for example, when different commercial firms, non-profit enterprises, program offices, and government agencies decide to share acquisition costs and risks in order to rapidly assemble, produce, deliver, or evolve innovative cybersecure modular OA software systems [Scacchi and Alspaugh 2015,Scacchi and Alspaugh 2016].

Our efforts are also aligned to Better Buying Power (BBP) initiatives [DoD 2016, Kendall 2015] to: (a) use Modular Open Systems Architectures to stimulate innovation; (b) strengthen cybersecurity throughout the (software system) acquisition life cycle [DoDGSA 2015]; and (c) increase the use of (OA software system) prototyping and experimentation. Beyond this, our investigation into smart contracts and associated technologies could contribute to new ways or means to specify or review acquisition contract incentives as well as tracking and improving contract performance.

Research Scope

There is a significant need for sustained research that investigates the interplay and inter-relationships between (a) current/emerging guidelines for the acquisition of software-intensive systems (including contract management and software development issues), and (b) how secure, reusable software product lines [Guertin, Sweeney, Schmidt 2015, Mactal, Spruill 2012, Womble, Schmidt et al. 2011] that employ a modular, cybersecure OA incorporating OSS/CSS component products (e.g., widgets, apps, and mashups) and their production processes [Scacchi and Alspaugh 2013b] are essential to stimulating innovation and improving the cost-reduction effectiveness of software system acquisition efforts.

Our acquisition research efforts are related to and primarily aligned with BBP initiatives that (a) use Modular Open Systems Architectures to stimulate innovation; (b) strengthen cybersecurity throughout the (software system) product life cycle; and (c) increase the use of (OA software system) prototyping and experimentation. As an example,

> "BBP 3.0 continues the emphasis on open systems architectures and modularity, focusing on *providing technical enablers and tools that can be employed by the acquisition workforce and industry* to enhance technology insertion, particularly in the most rapidly advancing areas of commercial technology (e.g. microelectronics, sensors, and software)...Such approaches should be considered for enabling competition for upgrades, facilitating reuse across the joint force, easing technology insertion, and aiding adoption of incrementally upgraded software" (emphasis added) [Kendall 2015].

Our research efforts address such concerns through cybersecure modular OA software systems that adhere to five principles: (a) Establish an agile, adaptive ecosystem environment for software component/system development and deployment; (b) Employ modular OA software system design and reference architectures that accommodate reuse of bespoke, licensed, or legacy software components; (c) Designate open interfaces for bespoke, licensed, or legacy OSS/CSS system components or subsystems; (d) Use open standards; and (e) Certify conformance to contractual, cybersecurity, and intellectual property requirements and customer rights. Research that advances the acquisition, production, and evolution of cybersecure modular OA software systems—especially those incorporating Web-based, mobile, or smart IoT devices—that follow these principles is highly relevant to for-profit industries and non-profit organizations, as well as to DoD and other government agencies.

Through our research, we seek to identify, track, and analyze acquisition costs, and development practices, for Web-based OA systems, mobile and emerging smart microelectronic IoT devices for use in enterprise software system applications. Such systems commonly integrate components independently developed by software producers using OSS or CSS, which then may be integrated into complete systems by system integrators [George, Morris, and O'Neil 2014, Reed, Benito, et al. 2012, Reed, Nankervis 2014, Scacchi and Alspaugh 2014]**.** Program managers, acquisition officers, and contract managers will increasingly be called on to review and approve cybersecurity measures employed during the design, integration, deployment, and evolution of OA systems [Scacchi and Alspaugh 2013b, Scacchi and Alspaugh 2013c].

Our research effort focused on performance of four concurrent research tasks. We briefly describe each research task then follow with an elaboration of our research description and the acquisition research questions we address.

We seek to identify, track, and analyze ways and means for how to articulate, tailor, and streamline the process for diverse acquisition scenarios for cybersecure modular OA software systems that accommodate Web-based, mobile, and smart IoT devices running software widgets, apps, and mashups. We seek to do so in ways that focus on software cost drivers and that highlight smart contracting opportunities for stimulating innovation that can realize cost reduction through modular cybersecure OA software components or system configurations. This investigation is therefore applicable to complex software elements used in many kinds of component-based OA software-intensive systems within government agencies, such as the DoD, as well as commercial firms and non-profit enterprises.

Realizing our research objectives and answering our research questions entails that our investigation focused on ***four research tasks*** in our approach, described in the next sub-section below. However, we propose that these four tasks are most effectively and most efficiently engaged when performed concurrently, rather than sequentially, due to the emergent nature of the proposed research line of study. Such concurrency also enables us to take advantage of advances in scientific knowledge or technological innovations that may appear during the course of our research efforts and task performance.

List of Research Tasks

i) Investigate the interactions between blockchains, smart contracts, and associated technologies with software system acquisition guidelines and processes, and the cost consequences of alternative software system architectures incorporating different mixes of OSS and CSS widgets, apps, mashups, and IoT device components subject to shared acquisition agreements among multiple parties that seek to produce assembled capabilities for C3CB applications using cybersecure modular OA components and SPLs [Scacchi and Alspaugh 2013a, Scacchi and Alspaugh 2013b, Scacchi and Alspaugh 2013c, Scacchi and Alspaugh

2015, Scacchi and Alspaugh 2016]. This entails exploring the balance between development, verification, and validation of software licenses and cybersecurity rights during procurement contract enactment, as well as the software widget, app, mashup, and IoT device component/license costs, while managing the development and evolution of OA systems at design-time, build-time, release and run-time, and post-deployment system evolution.

ii) Develop and/or refine formal foundations for establishing acquisition guidelines, blockchain and smart contracting practices that program managers can use in diverse acquisition scenarios for reduced cost software-intensive systems that rely on development and deployment of secure modular OA systems using OSS widgets, apps, mashups, and IoT devices, as well as SPL technology and processes [Scacchi and Alspaugh 2011, Scacchi and Alspaugh 2012, Scacchi and Alspaugh 2013a, Scacchi and Alspaugh 2013b, Scacchi and Alspaugh 2013c, Scacchi and Alspaugh 2014, Scacchi and Alspaugh 2015, Scacchi and Alspaugh 2016].

iii) Continue to develop concepts contributing to the emerging design of an automated approach supporting acquisition of cybersecure, modular OA software systems by (a) determining their conformance to acquisition guidelines/policies, contracts, and related license management issues, and (b) giving future acquisition workforce support and insights to properly review, approve, and manage the acquisition of complex systems that incorporate cost-sensitive acquisition of cybersecure OA systems composed from software widget/app or software-based IoT device components [Scacchi and Alspaugh 2011, Scacchi and Alspaugh 2012, Scacchi and Alspaugh 2013a, Scacchi and Alspaugh 2013b, Scacchi and Alspaugh 2013c, Scacchi and Alspaugh 2014, Scacchi and Alspaugh 2015, Scacchi and Alspaugh 2016].

iv) Document 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; (b) a research paper to be presented at the *14th Annual Acquisition Research Conference*, in Monterey, CA, May 2017; (c) a progress report with the OSD sponsor via a video teleconference or other meetings at a time to be determined during the period of the award; and (d) related research venues and publications, including periodic progress reports.

Relevance of Our Efforts to Acquisition Research and Practice

Overall, through this research effort, we continue to seek to identify, track, and analyze ways and means for how to articulate, tailor, and streamline the process for diverse acquisition scenarios for secure OA systems through use of blockchains and smart contracts that accommodate OA system supply chains that deliver Web-based and mobile devices running widgets, apps, and mashups. We seek to do so in ways that focus on innovative opportunities emerging from the potential introduction of blockchains and smart contracts in OA system acquisition processes and ecosystems. This investigation is therefore applicable to complex software elements used in many kinds of component-based OA software-intensive systems within business and academic enterprises, other non-governmental organizations, as well as DoD and other governmental organizations. Furthermore, through these four tasks, this acquisition research supports and advances a public purpose by investigating challenges arising from the adoption and deployment of, which is a broad audience for our research [Scacchi and Alspaugh 2014b, Scacchi and Alspaugh 2014c, Scacchi and Alspaugh 2015, Scacchi and Alspaugh 2016, Scacchi and Alspaugh 2017a, Scacchi and Alspaugh 2017b].

Finally, we note that academic institutions, government agencies, and most large-scale business enterprises continually seek new ways to improve the functional capabilities of their software- intensive systems through lower acquisition costs. The acquisition of OA systems that can adapt and evolve through replacement of functionally similar Web-based and mobile device-based software components and SPLs is an innovation that can lead to lower cost systems with more powerful, more agile functional capabilities. There is a significant need for sustained research that investigates the interplay and inter-relationships between (a) current/emerging guidelines for the acquisition of software-intensive systems, and (b) how secure, reusable software product lines

[Mactal and Spruill 2012, Womble, Schmidt, Arendt, Fain 2011] that employ an OA incorporating OSS/CSS component products (e.g., widgets, apps, and mashups) and their production processes [Scacchi and Alspaugh 2013b], are essential to improving the buying power and cost-reduction effectiveness of software-intensive program acquisition efforts.

OA system acquisition, development and deployment are thus an approach to realizing better buying outcomes for lowering system costs while jointly enabling more competition through the adoption of OA systems that utilize standardized interfaces, utilize OSS components where appropriate, increase small business roles and opportunities, use of technical development phase for true risk reduction and rapid prototyping, as well as doing more without more [Scacchi and Alspaugh 2014a, Scacchi and Alspaugh 2015].

Last, we are grateful for the support and funding we have received that enabled our acquisition research to continue, and as documented in this Final Report. We welcome any comments or questions regarding any materials or concepts presented in this Report.

THIS PAGE INTENTIONALLY LEFT BLANK

Cybersecure Modular Open Architecture Software Systems for Stimulating Innovation

How might we stimulate the development of innovative approaches to continuously assuring the cybersecurity of Open Architecture (OA) software system? This is the acquisition research challenge we are addressing. In particular, we are interesting in investigating innovations that represent either incremental improvements or substantial departures in current acquisition practice of such systems. We target our efforts to practical OA software system production, deployment and sustainment for applications like command and control, or business enterprise (C2/B) systems that are central to the mission and operations of military or industrial enterprises. So we seek to stimulate significant innovations that employ emerging concepts and technologies to problems observable with the acquisition, development, and evolution of modern C2/B systems.

Our interest is to stimulate the development of innovative approaches to continuously assuring the cybersecurity of Open Architecture (OA) software system. We focus attention to exploring the potential for using blockchains and smart contract techniques, and how they can be applied to support acquisition efforts for software systems for OA command and control, or business enterprise (C2/B) systems. We further limit our focus to examining the routine software system updates to OA software configuration specifications that arise during the development and evolution processes arising during system acquisition. We find that there are new ways and means by which blockchains and smart contracts can be used to continuously assure the cybersecurity of software updates arising during OA software system development and evolution processes. We present a case study examining software evolution process that updates an OA C2/B system, to describe these details. We then discuss some consequences that follow for what emerges from these innovations in expanding the scope of cybersecurity assurance of not just the delivered OA C2/B software systems, but to the engineering processes which create, transform or otherwise update technical data that is central to the acquisition of OA software systems.

Problem

The particular problem we investigate here is how best to develop and demonstrate a new conceptual approach to providing continuous cybersecurity assurance [cf. DoDGSA 2013] with OA C2/B software systems in response to evolutionary updates to currently installed software configurations that routinely arise during the technical development and maintenance, upkeep, and sustainment in the field—what we call, software evolution [Scacchi and Alspaugh 2012, Scacchi and Alspaugh 2017a].

Solution

The innovation we focus our attention to are the concepts, techniques, and technologies that denote blockchains and smart contracts, along with how they can be used to continuously assure the cybersecurity of software updates arising during OA software system development and evolution processes that span software supply chains.

Approach

Our efforts focus on an innovative utilization of blockchains and smart contracts within the technical software development and evolution processes that arise within the acquisition of complex, OA C2/B software systems. We are not focusing attention at this time to software purchasing activities or financial transactions, though blockchains and smart contracts are likely to stimulate innovations in this aspect of OA software system acquisition.

Why this approach?

Based on prior studies of issues and challenges arising in the development and evolution of OA software systems for C2/B system applications [Guertin, Sweeney, Schmidt, 2015, Scacchi and Alspaugh 2012-2017, Womble, Schmidt, Arendt, Fain 2011], we have already drawn attention to technical problems that arise in the software engineering processes that software producers, system integrators, and customer end-users (both enterprises and individuals therein) experience. But we recognize these processes are partially-ordered sets of activities whose completion often entails technical data transactions like creation of digital system design documents, composition and integration of software components (e.g., applications, mobile apps, plug-in widgets), and deployed software executable/update packages that are stored, installed, and tracked in different online repositories across a network environment. At present, these transactions often lack a common or centralized repository for tracking these diverse transactions across networked platforms that span an OA software system ecosystem (a supply chain network from producers to system integrators to customer enterprises/individuals). We believe blockchains are a candidate for this. These transactions similarly lack a common and potentially reusable specification for how to manage and track such software engineering transactions in forms that are open to independent validation and audit. We believe smart contracts are a candidate to address this.

THIS PAGE INTENTIONALLY LEFT BLANK

Background: Blockchains, Smart Contracts and Software Supply Chains

Blockchains

Blockchains are a 21st Century computational mechanism for realizing the equivalent of the traditional bookkeeping ledger utilized in finance and accounting. Such ledgers record and track the assignment of incoming (budget authorization or revenue) and outgoing (allocations and expenses) enterprise transactions and denominated amounts, whether in a monetary currency, bartered trade, or some other transactional resource (e.g., gold bullion, Bitcoins, original artworks) [DuPont and Maurer 2015]. Such transactions are grouped in blocks, for example a set of interrelated OA software system updates may be grouped together into a block that denotes a transformation of the current system configuration into an evolved system configuration.

Figure 1 shows a model for what a small blockchain may look like if visualized as a chain-like structure.

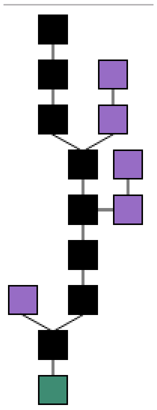
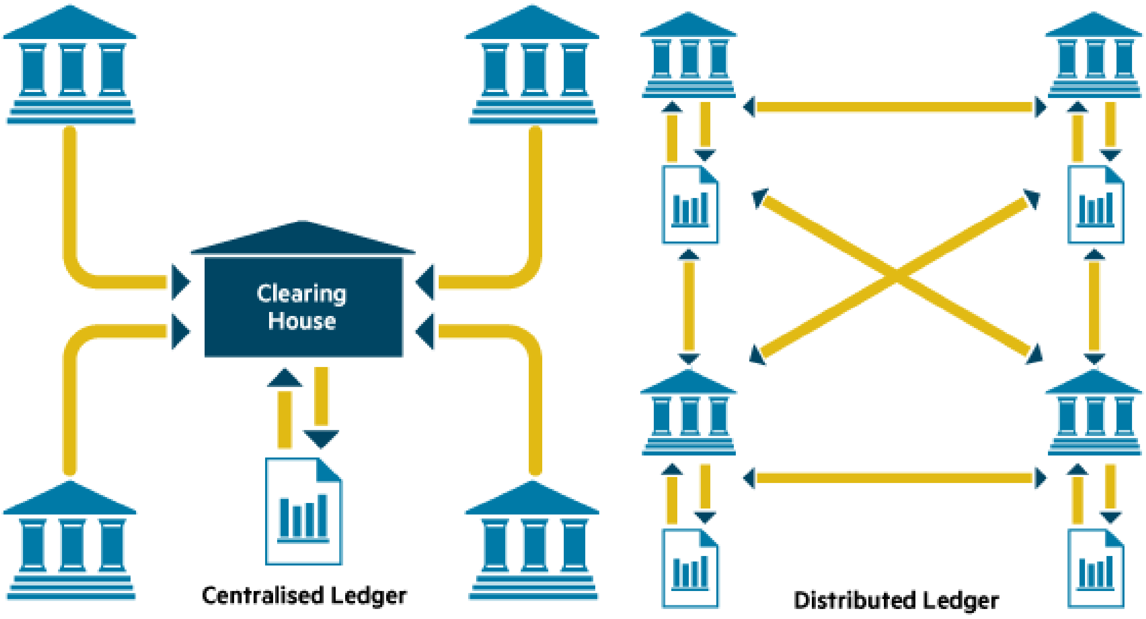


Figure 1. Visual model of a simple blockchain, highlighting individual update transaction blocks (squares), as well as the longest path of validated update transactions. Transaction blocks not on the longest path are considered as no-longer-valid elements of the blockchain.

Both transactions and blocks are serialized, logged, timestamped, and tracked in ways that are open to internal, external or independent verification and audit by decentralized third-parties [Blockchain 2017]. Updates to the blockchain are allowed only by consensus of remote mechanisms and proofs of work by anonymous, untrusted service providers (called miners) who collect a modest execution fee for their efforts. The payment and deposit of an execution fee also mitigates against the actions of unknown others who might act to corrupt the blockchain state. Finally, blockchains can be realized as persistent databases or cloud-based repositories [Blockchain 2017]. Such repositories might be utilized, for example, to record and store a bill of materials detailing all software elements that are composed into a specified software system configuration, as well as the itemized serialization of the evolutionary updates to any of the software elements therein, across the development and maintenance life cycle of an OA software system [cf. Scacchi and Alspaugh 2017a]. Figure 2 displays a traditional centralized ledger versus a decentralized blockchain ledger.

Embedding distributed ledger technology

A distributed ledger is a network that records ownership through a shared registry



In contrast to today's networks, distributed ledgers eliminate the need for central authorities to certify ownership and clear transactions. They can be open, verifying anonymous actors in the network, or they can be closed and require actors in the network to be already identified. The best known existing use for the distributed ledger is the cryptocurrency Bitcoin

FT graphic. Source: Santander InnoVentures, Oliver Wyman & Anthemis Partners

Figure 2: Traditional ledger network on left, decentralized blockchain ledger network on right.

Blockchains operate as an append-only data structure or database maintained by a decentralized collection of mutually distrusting computational nodes participating in a peer-to-peer network. Blockchains are secure by design [Blockchain 2017]. Blockchain ledgers are updated (appended) as a result of recorded transactions, much like a personal bank account is updated through deposit, withdrawal, credit or debit transactions made by the account holder, through a third-party (the bank or transaction system processor), who may charge a fee for transactions. Much like bank account transactions, blockchain update transactions are distributed over a network, time-stamped, persistent, and verifiable. However, the peer-to-peer network of blockchain nodes is a decentralized autonomous authority without legal standing, compared to the centralized authority taken by a bank or credit/debit card transaction processor.

Smart Contracts

Smart contracts denote the computational counterparts of traditional paper contracts for how a group of interrelated transactions will be governed to assure fulfillment of terms, conditions, rights and obligations. Within distributed ledger applications and blockchain associated technologies, smart contracts are denoted by software programs that can be automatically executed whenever blockchain transactions occur. Such transactions, for example, may be associated with the acquisition of a complex system or with the ongoing procurement of retail supply purchasing agreements. These smart contracts denote networked software system protocols that facilitate, verify, or enforce the negotiation or performance of a specified contract, and thus which transactions to process (where, when, how, and for what parties) in what order [Smart Contracts 2017]. They are realized using computer based, formal specifications of transaction-based processes that can be codified into executable computer programs. Such computational support allows for modeling, analysis and simulation of transactions or processes that can be enacted, verified and validated at Internet-time speeds, with precision and automated recall of transaction details well beyond what enterprises traditionally have performed. Smart contracts also allow for the establishment and operation of decentralized autonomous services that allow for cooperating parties to enact and fulfill the details of a shared contract through just automated means. Next, smart contracts are automatically enforced by the consensus mechanism associated with the blockchain. Smart contracts are thus attractive to use to securely manage recurring transactions between known or unknown parties, such as those associated with updating the technical data, source code, repositories, and related artifacts associated with software development and evolution processes associated with large, long-term software acquisition efforts.

Software Supply Chains and Ecosystems

Software elements and configured system are developed by component or system producers on the way to being adopted and deployed by customer organizations or end-users. Many times, the software elements are subjected to value-added system integration efforts which expand the scope and functional capabilities of the resulting integrated system for deployment, or may otherwise integrate these elements within legacy installed software systems. This ecosystem of producers, system integrators, and customers form a network of relationships that is commonly called a *software supply network.* Such a network may offer many possible pathways that enable the flow of newly produced or integrated software elements (e.g., new software products, apps, or widgets) in particular configurations that are targeted to a specific type of software deployment platform, installation or ecosystem niche [Scacchi and Alspaugh 2012]. Such a path from producers through integrators to customers denotes a specific *software supply chain*. A generalized abstract depiction of a software ecosystem as a software supply network is shown in Figure 4.

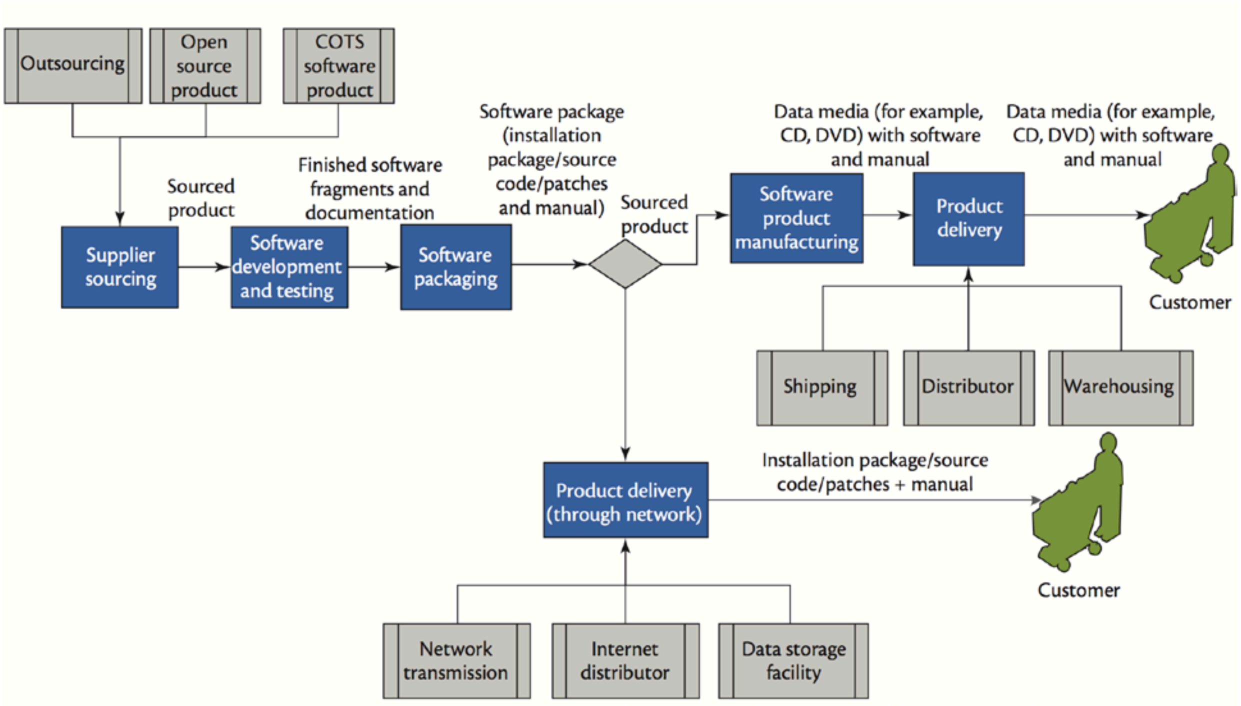

Outsourcing
Open source product
COTS software product
Sourced product
Finished software fragments and documentation
Software package (installation package/source code/patches and manual)
Sourced product
Data media (for example, CD, DVD) with software and manual
Data media (for example, CD, DVD) with software and manual
Supplier sourcing
Software development and testing
Software packaging
Software product manufacturing
Product delivery
Customer
Shipping
Distributor
Warehousing
Installation package/source code/patches + manual
Product delivery (through network)
Customer
Network transmission
Internet distributor
Data storage facility

Figure 3. Software supply chain development processes [Al Sabbagh & Kowalski 2015].

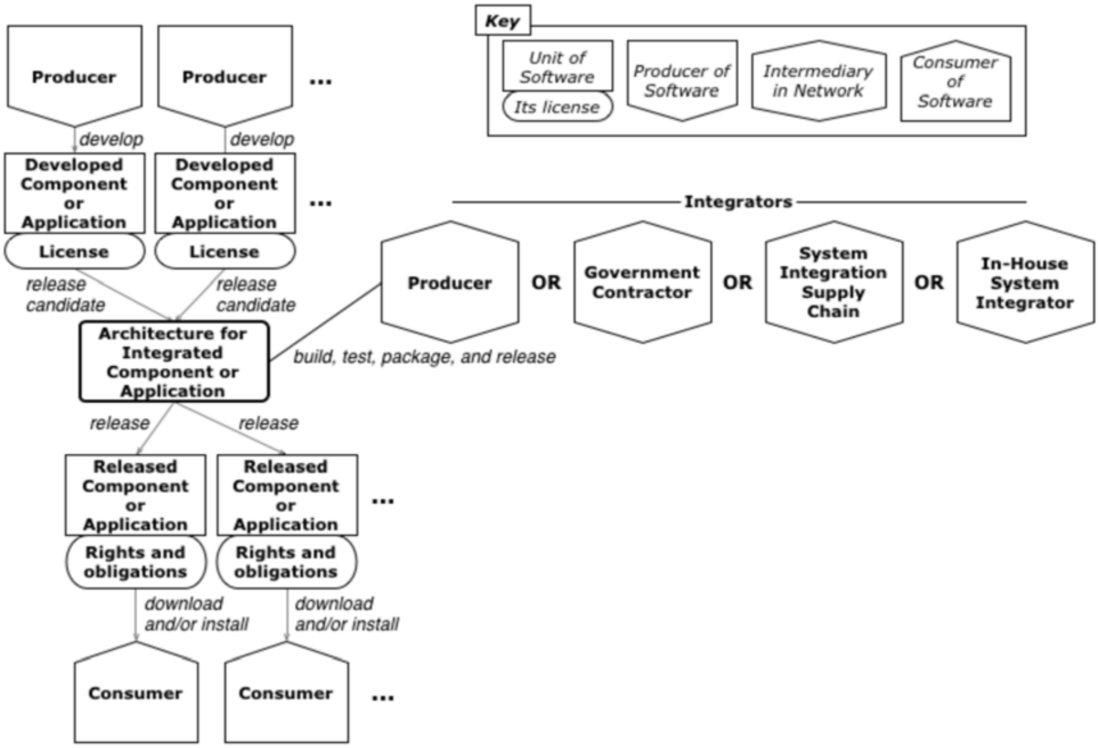


Figure 4. An ecosystem model of a software supply network connecting software producers, system integrators and consumers (customers/end-users) [cf. Scacchi and Alspaugh 2012, Scacchi and Alspaugh 2017a].

A small number of software supply chain researchers has sought to develop models for software supply chains that can be visually rendered to aid in facilitating understanding and communication. For example, researchers at the Software Engineering Institute have sought to visualize software supply chains as directed graphs [Ellison, et al 2010]. But such a model may somewhat obscure how software elements move through a development process life cycle, especially when iterative development processes are employed. Alternatively, others like Al Sabbagh & Kowalski [2015] draw attention to explicit development process flows, as shown in Figure 3. Such a representation can also incorporate annotations for denoting where different kinds of social or technical risks to the integrity of the software supply chain may arise, which provides both foundational and practical insights to which processes maybe subject to different types of cybersecurity threats. Unfortunately, such a process-centered rendering slights inter-relationships between different participating producers, system integrators and customers.

Our view of a software ecosystem seeks to combine or unify these alternative approaches to modeling and visualizing software supply chains, as have appeared in our earlier efforts [Scacchi and Alspaugh 2012, Scacchi and Alspaugh 2016, Scacchi and Alspaugh 2017a]. However, we note that at this point we do not have a single visual representation that combines a software supply network and process flow into a single rendering that may be complex and thus obscure, but instead rely on multiple visualizations, one for the supply network, and others for OA software configurations that result from different software development or evolution processes. Said more simply, consider integrating the view from Figure 3 in place of the “Architecture for Integrated Component or Application” box in the following Figure 4. That is, one or more OA software system “Integrators” routinely enact software supply chain development processes [Scacchi and Alspaugh 2013b, Scacchi and Alspaugh 2017a].

The software ecosystem schema in Figure 4 represents the ecosystem immediately connected to a particular component/application integrator. It is recursive: a “producer” in the top row of the figure can itself be a software ecosystem, determined by its own architecture and supplied by its own producers, for which the consumer is the architecture and integrator of the larger schema's product, and a "consumer" in the bottom row may be a software ecosystem for a further component/application. The basic steps of the recursion in each direction are (i) a consumer who is a user only, and (ii) a producer of a simple component, one that is developed from scratch and makes use of no subcomponents. A typical large OA system will contain a number of such simple components that act as shims or scripts between larger components with related but nonidentical interfaces.

Blockchains are being extended to accommodate smart contracts that allow for the formation of virtual, decentralized autonomous organizations (DAOs) [Ethereum 2017] that can span diverse software ecosystems of different size, connectivity, and complexity. DAO in turn can be designed to govern, enforce, and assure the integrity and validity of complex or idiosyncratic blockchain update transactions on supply chains of different types [Smart Contracts 2017]. In our case, a software supply chain delivers software elements or systems through a DAO that denotes a given configuration of participating software producers, system integrators and customer organizations.

Our interest is focused on software supply chains that enable continuous flow of developmental or evolutionary updates to software elements configured to operate within an OA software system. In one acquisition scenario, this might entail the procurement of pre-certified and secured software apps from a secured, online app/component store [George, Galdorisi, et al. 2014, George, Morris, et al. 2014]. Alternatively, multiple independent Program Offices or independent enterprises may seek to partner with other parties to share the cost of developing bespoke OA software system apps or components. Figure 5 presents a notional depiction of two alternative acquisition scenarios. Overall, multi-party agreements for coordinated system acquisition can denote a kind of DAO whereby two or more Program Offices or other enterprises can act to shared the procurement costs of a new C2/B system application or component, of mutual interest to the participating parties [cf. Reed, Benito, Collens, Stein, 2012, Reed, Nankervis, Cochran, Parekh, Stein, 2014, Scacchi and Alspaugh 2015].

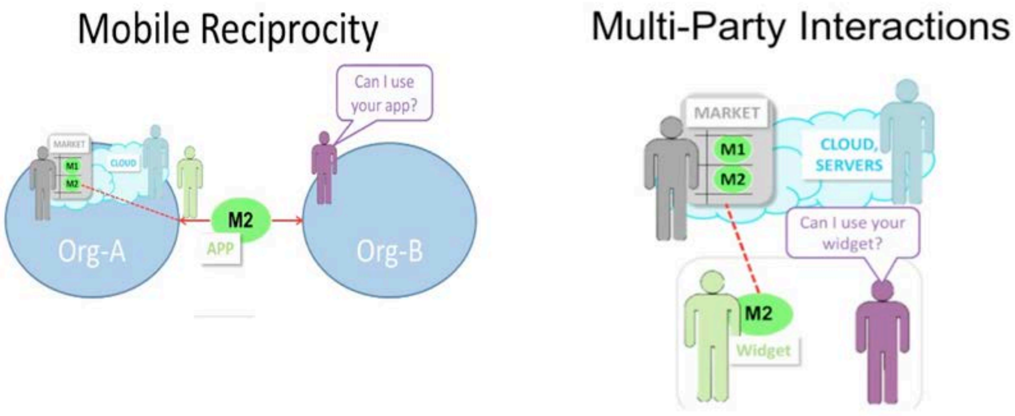


Figure 5. Notional depiction of two alternative scenarios entailing “mobile reciprocity” or “multi-party interactions” for acquisition of OA software system components, applications or integrated system capabilities [Reed, Benito, Collens, Stein, 2012, Reed, Nankervis, Cochran, Parekh, Stein, 2014, Scacchi and Alspaugh 2015].

No single participating Program Office or acquisition enterprise necessarily is “in charge” of the overall shared acquisition, so all parties participate on the basis of their ability or resources they contribute to realize the shared goal of a particular DAO. Similarly, smart contracts can govern transactions between mutually distrusting participants that are automatically enforced by automated consensus mechanisms associated with blockchain updates. This capability thus provides a mechanism for detecting, rejecting or preventing unauthorized update transactions to the blockchain, as might be attempted via a cyber attack during OA software system development or evolution. Accordingly, our interest is to investigate how blockchains, smart contracts and related technologies can be utilized to improve cybersecurity, specifically to manage and track software engineering development and evolution processes that entail process transactions that update the configuration of OA software systems.

So how might we utilize blockchains and smart contracts to innovate the continuous development and evolution of OA systems? How can this be conceived and applied in ways that are not specifically limited to financial transactions commonly associated with system acquisition? Before we can answer such questions, we need to more closely examine what kinds of cybersecurity threats to software supply chains we want to defend against using blockchains and more.

Social, Technical, and Unintentional Cybersecurity Threats to Software Supply Chains

Recent Cybersecurity Attacks on Software Supply Chains

Coordinated international attacks on vulnerable software-intensive systems of high value and controlling complex systems are becoming ever more apparent. Security threats to software systems are multi-valent, multi-modal, and distributed across independently developed software system components. Five recent attacks illustrate these characteristics, and also a progression from attacks against systems, to attacks against ecosystems that result in compromised systems developed in those ecosystems, to attacks against tools used in multiple ecosystems that result in compromised systems developed in any of those ecosystems. Consider the following recent events that demonstrate different ways and means through which software supply chains have become vulnerable to cybersecurity attacks.

Stuxnet: malicious code distributed via physical storage devices across a virtual software supply chain

Stuxnet [Falliere, Murchu, Chien 2011] was a well planned attack of cyberphysical systems used to control industrial system operations, including those associated with nuclear materials processing. Stuxnet was discovered in July 2010, but subsequent analysis indicated that thousands of industrial control system software worldwide were eventually infected and subject to cybersecurity attacks that utilized vulnerabilities exploited by Stuxnet. Stuxnet is thus a significant example of a successful attack on an poorly perceived software supply chain—the virtual network of mostly unconnected computers running the targeted software systems (e.g., Supervisory Control and Data Acquisition (SCADA) software package). The coordinated Stuxnet attack employed a bundle of attack/viral vectors and social engineering tactics in order for the attack to reach strategic industrial control systems that were isolated and walled off ("air gapped") 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 its goal was to go outside or beneath its entry context. Furthermore, 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.

NotPetya: malicious code distributed through access to producer source code

NotPetya, discovered in June 2017, used the Ukrainian tax accounting software M.E.Doc as an infection vector. Once installed, it attempts to propagate across a network using any of four different exploits. It collects usernames, passwords, and other confidential information and installs a backdoor giving the attackers control of the machine. If commanded by the attackers, it destructively encrypts files and, if administrator privileges are obtained, the master boot record, rendering the files irretrievable and the computer unusable; this part of the attack masquerades as ransomware but victims who pay ransom apparently do so fruitlessly. The attackers injected a backdoor into a legitimate M.E.Doc module, presumably with access to the source code; the compromised class methods were then invoked when the software checked for updates [Cherepanov 2017, US-CERT 2017].

CCleaner: attack performed upstream of cryptographic signature and distribution

A version of the widely-used and free CCleaner utility that was downloaded in August and September 2017 was found to contain a backdoor with remote system administration tools. The malware was piggybacked on valid CCleaner releases on legitimate download servers, and cryptographically signed with a valid certificate issued to Piriform, CCleaner's producer. It appears that attackers gained access to at least part of the CCleaner development or build environment [Brumaghin, Gibb, et al. 2017, Menn 2017].

XCodeGhost: infected developer tools infect software they create

In late 2015, a fake version of Apple's XCode development tools was placed on unofficial sites for Chinese developers. The fake XCode tools injected

XCodeGhost malware into apps developed using them, thus infecting supply chains of software to which the attackers need not have access [Greenberg 2017].

Equifax: through a bug in widely used open source software component, inserting persistent attack software and remote control of enterprise systems, enabling prolonged systematic data exfiltration

In Spring 2017, Equifax, one of the three leading credit-reporting firms in the U.S. was exposed to a remote attack through a known technical vulnerability in its backbone software infrastructure (e.g., *Apache Struts*). As knowledge of the Struts vulnerability and its software update repair was publicly disclosed, it was clear that resolving this problem requires a concerted software update effort in any organizational or infrastructural system configuration where it was installed. A simple, pre-coded software patch was not available, nor was it appropriate, due to the configurable data processing capabilities that Struts provides. While the social and technical details of the Equifax breach are described in greater detail elsewhere [Riley, Robertson, Sharpe 2017], it also appears that the attack was prolonged due to unintentional conditions and events arising from contractual disputes between Equifax and its third-party cybersecurity service provider regarding the efficacy of contracted service performance. As a consequence of the threats and unintentional conditions, the attack persisted for months, and that dozens if not more unauthorized software updates to installed software configurations on different Equifax enterprise systems were propagated across Equifax networks and multiple databases. Remotely controlled system and data analysis tools were covertly installed that could query accessible data assets to reveal their contents, as well as install other secondary software tools that could covertly extract and encrypt appropriated data, then disseminate gigabytes of acquired data over public networks over combinatorially diverse paths (e.g., darknet torrents) to hidden/masked destinations in other countries. The scale, sophistication, and continued covert software installation suggests a state-sponsored attack enterprise utilizing multi-mode entry and attack vectors, much like *Stuxnet*, rather than an individual or simple criminal endeavor [Riley, Robertson, Sharpe 2017].

Social and Technical Threats to OA Software Supply Chains

More generally, cybersecurity threat categories can be identified starting from an interpretation of Wang et al. (2013) that is augmented with other constructs or concepts from secure open architecture software systems found in papers by Scacchi and Alspaugh [2008-2017]. The security threat meta-model identified below is grouped into three sections, each beginning on a new page for clarity. This is followed by diagrams and excerpts from the threat model by Wang et al. [2013]. The social threats and technical threats from Wang et al. have been modified and expanded to accommodate our concepts, and thus serve as a basis for developing a security threat meta-model for open architecture software systems. What is needed is an articulation of a *security threat meta-model* that incorporates concepts, constructs, tools, and capabilities derived from blockchains, smart contracts, software taggants, and *smarter contracts* (smart contracts that stipulate enactable software security license obligations and rights).

Following Al Sabbagh and Kowalski (2015), software security countermeasures need to address, for example, social threats when recipients of a software product deny receiving it, a social countermeasure would be to legally require a third-party notary (e.g., blockchain miner) to prove that recipients actually received the software product (i.e., verification and non-repudiation of update transaction). A technical countermeasure to deal with the same threat would be the implementation of digital signatures using public-key cryptography (cf. software taggants [Kennedy and Muttik 2011]). Another example of using counter-measures is thwarting the threat of malicious code being injected into source code while transmitted over the network. A social countermeasure would be implementation of a third-party escrow (via blockchain), where a technical countermeasure would be implementation of virtual private networks (blockchains). In future research effort, we envision cybersecurity countermeasures (can be formally specifically) using OA software system security licenses that are computationally enacted through smart contracts that stipulate defensive, detective, or preventive cybersecurity countermeasures.

We provide an expanded and revised list of 30 social threats and 24 technical threats informed by similar lists presented in Wang, Al Sabbagh, and Kowalski (2013).

Social Threats to Software Supply Chains

ST1: Supplier of software product denies having sent the software product.

ST2: Ordered software products such as outsourced software components could not arrive on time because of non-technical reasons such as delivery mistake.

ST3: Secret information (ex. hard-coded key, seed value) about the outsourced software component is disclosed unintentionally by internal employee.

ST4: Like ST3, but intentionally because of bribery or some other reason.

ST5: Unauthorized people get access to the secret information of outsourced software component through non-technical reasons such as spoofing.

ST6: Security weakness information about the sourced software product is disclosed unintentionally by internal employee to unauthorized people.

ST7: Like ST6, but intentionally.

ST8: Unauthorized people get access to the security weakness information of outsourced software component through non-technical reasons such as spoofing.

ST9: Secret information (ex. hard-coded key, seed value) about the software product is disclosed unintentionally by internal employee to unauthorized people.

ST10: Like ST9, but intentionally.

ST11: Unauthorized people get access to the secret information of the software product through non-technical reasons such as spoofing.

ST12: Security weakness information about the software product is disclosed unintentionally by internal employee to unauthorized people.

ST13: Like ST12, but intentionally.

ST14: Unauthorized people get access to the security weakness information through non-technical reasons such as spoofing.

ST15: Source code or installation package is destroyed unintentionally by internal employee for authorized people.

ST16: Like ST15, but intentionally.

ST17: Malicious code is inserted into the source code or installation package unintentionally by internal employee.

ST18: Like ST17, but intentionally by unauthorized people.

ST19: Unauthorized people get access to the source code or installation package, modify it or destroy it (NB: “evolution update transactions”) through non-technical reasons such as spoofing.

ST20: (Un)authorized people can/cannot get access to the source code or installation package because of non-technical reasons such as flooding.

ST21: Data storage facility for source code or installation package becomes unavailable to Unauthorized people because of non-technical reasons such as flooding.

ST22: User guide of the software product is modified or deleted (NB: “evolution update transactions”) unintentionally by internal employee.

ST23: Like ST22, but intentionally.

ST24: Unauthorized people get access to the user guide of the software product, modify it or destroy it (i.e., these are unauthorized OA software “evolution update transactions”) through non-technical reasons such as spoofing.

ST25: Cannot get access to the user guide of the software product because of non-technical reasons such as flooding.

ST26: Data storage facility for user guide becomes unavailable because of non-technical reasons such as flooding.

ST27: Real software products are replaced by counterfeit (NB: “evolution update transactions”)

ST28: Recipient (customer or staff working in the delivery process) denies the receipt of the software product.

ST29: Internal employee destroys data media unintentionally.

ST30: Security mechanism (ex. length of the key) deployed within the software product is not allowed by the applicable law of the end-customer.

Technical Threats to Software Supply Chains

TT1: Malicious code is inserted into open source tool by unauthorized people through technical approach, which leads to security defects of the software product.

TT2: Secret information (ex. hard-coded key, seed value) about the outsourced software component is obtained by unauthorized people through technical approach such as hacking.

TT3: Unauthorized people get access to the security weakness information of outsourced software component through technical approach such as hacking.

TT4: Unauthorized people insert malicious code into the outsourced software component while it is in storage using technical approach such as hacking.

TT5: Malicious code is inserted into the outsourced software component by unauthorized people during delivery through technical approach.

TT6: Outsourced software component is destroyed by unauthorized people through technical approach such as attacking the storage facility.

TT7: Unauthorized people get access to the security weakness information of the software product through technical approach such as hacking.

TT8: Secret information (ex. hard-coded key, seed value) about the software product is obtained by unauthorized people using technical approach such as hacking.

TT9: Malicious code is inserted by unauthorized people into the source code or installation package when it is in storage through technical approach such as hacking.

TT10: Malicious code is inserted into source code or installation package when it is stored in the data media by unauthorized people during product delivery (physical delivery).

TT11: Malicious code is inserted into source code or installation package of the software product during network transmission by unauthorized people through technical approach.

TT12: Source code or installation package is destroyed by unauthorized people when it is in storage through technical approach.

TT13: Source code or installation package is destroyed by unauthorized people during network transmission through technical approach.

TT14: Network access to the source code or installation package is destroyed by attackers using technical approach such as DOS attack.

TT15: Data media is destroyed by unauthorized people through technical approach.

TT16: Unauthorized people get access to the user guide of the software product, and modified it intentionally through technical approach such as hacking.

TT17: Unauthorized people get access to the user guide of the software product, delete it or modify it to make it unavailable through technical approach such as hacking.

TT18: User guide of the software product is damaged or modified during network transmission through technical approach such as hacking during network transmission.

TT19: User guide of the software product is modified through technical approach such as session hijacking attack during network transmission.

TT20: Malicious code is inserted into patches by unauthorized people when it is in storage through technical approach such as hacking.

TT21: Malicious code is inserted into patches during network transmission through technical approach such as hijacking by unauthorized people.

TT22: Patches are destroyed by unauthorized people when it is in storage through technical approach.

TT23: Patches are destroyed by unauthorized people during network transmission through technical approach.

TT24: Network access to the patches is destroyed by unauthorized people using technical approach such as DOS attack.

Finally, Al Sabbagh and Kowalski [2015, Also see Wang, et al. 2013] provide a visual model that seeks to associate where cybersecurity threats such as those identified above may arise within different software development processes that span software supply chains. Their model is shown in Figure 6 below.

Other Unintentional Socio-Technical Threats

Both social threats and technical threats identified above are amenable to intervention, detection, or prevention via different kinds of cybersecurity mechanisms or practices. However, there are also other *unintentional socio-technical threats* that emerge through unexpected acts, conditions, or events identified as: mistakes, errors, breakdowns, accidents, glitches, anomalous events, system outages, system failures, system implementation failures, and the like. Such acts, conditions, or event can create externalities or effects that temporarily defeat, reset, or bypass cybersecurity system elements, configurations, or settings whose normal operation can provide effective cybersecurity protections or assurances. Here we identify examples of these threats.

– *Mistakes* may arise, for example, in mis-entering the values for the configuration or update of configuration information used to assign security protections following a Security Technical Implementation Guide (STIG).

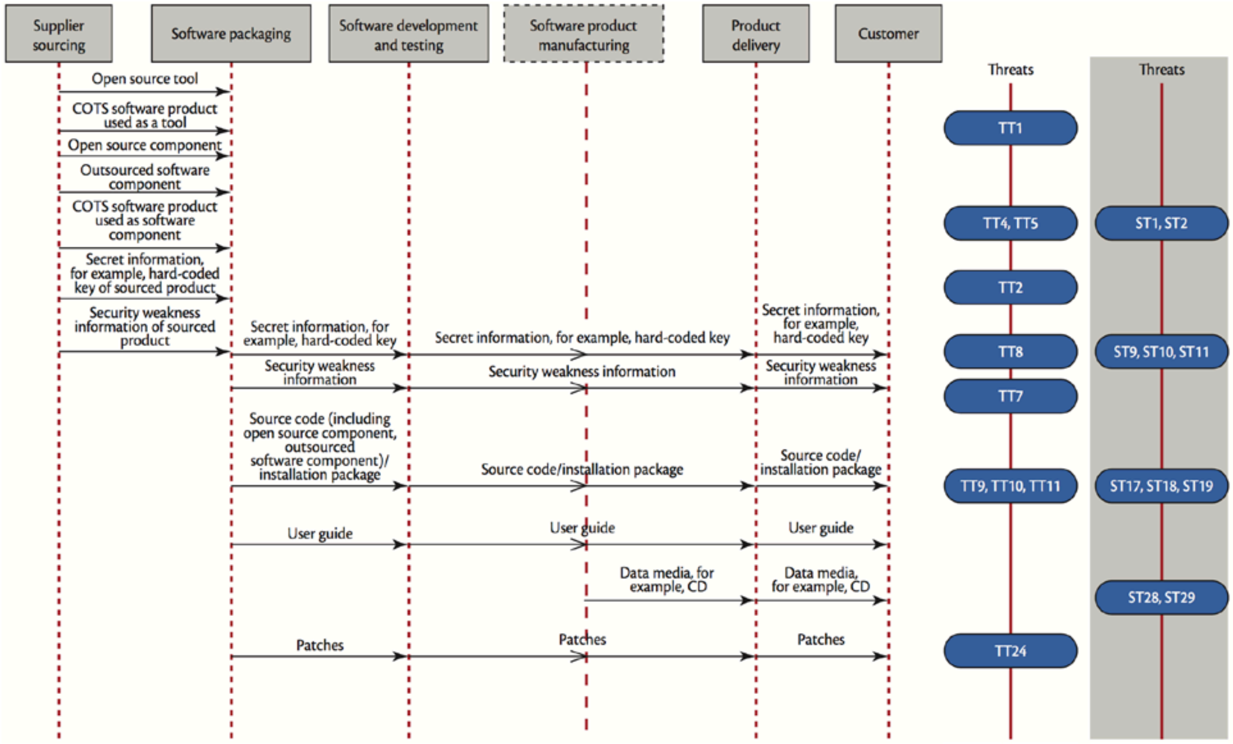

Supplier sourcing
Software packaging
Software development and testing
Software product manufacturing
Product delivery
Customer
Threats
Threats
Open source tool
COTS software product used as a tool
Open source component
Outsourced software component
COTS software product used as software component
Secret information, for example, hard-coded key of sourced product
Security weakness information of sourced product
Secret information, for example, hard-coded key
Secret information, for example, hard-coded key
Secret information, for example, hard-coded key
Security weakness information
Security weakness information
Security weakness information
Source code (including open source component, outsourced software component)/ installation package
Source code/installation package
Source code/ installation package
User guide
User guide
User guide
Data media, for example, CD
Data media, for example, CD
Patches
Patches
Patches
TT1
TT4, TT5
TT2
TT8
TT7
TT9, TT10, TT11
TT24
ST1, ST2
ST9, ST10, ST11
ST17, ST18, ST19
ST28, ST29

Figure 6. Al Sabbagh and Kowalski [2015] model for associating selected social and technical threats identified earlier with software development processes spanning software supply chains.

– *Errors* may arise due to omission, commission, or miscalculation with omission errors resulting from security values that are not entered or updated per guidelines (e.g., ignoring instructions to reset the default password, or entering an easily guessed very weak password). Errors of commission may denote those arising from inadequate training of proper system operating conditions and user-system interactions (e.g., “I didn’t know I was suppose to do that” or “I didn’t read the manual”). Commission errors may also arise due to other forms of incompetent system use, or where software system designers assume system users have certain skills or think/act is certain ways regarding proper system use, which turns out not to be the case. Errors of miscalculation entail usage conditions where users mistype or mis-enter data, code, or formulae that otherwise look correct as plausible input, which in turn may give rise to downstream calculations or outputs that modify system data/operations, that in turn precipitate other unintentional conditions, events, or actions.

– *Breakdowns* can arise at any time when a workflow utilizes a software component or application that for some reason, does not operate as expected, freezes, “hangs” or suddenly stops and exits without warning, such that the status/state of the attendant work-in-progress in unclear, garbled, or lost. Breakdowns thus require some form or rework to recover from the breakdown.

– *Accidents* can take many forms, but include matters such as blunt object falls/drops, or beverage/food/liquid spills, on computer keyboards or desktop peripherals (e.g., removable disk drives) which may introduce electrical short circuits that are misinterpreted by the computer as a user input or command sequence invocation, that in turn may undo or bypass currently active security system elements.

– *Glitches* are peculiar system behaviors that often denote hidden/latent computational concurrencies that give rise to conditions like deadlocks, mutual exclusion race conditions, infinite loops, memory leaks or spillover effects. Rectifying such glitches often entails activities like restarting or shutdown-and-startup the computer system, but without any knowledge of whether any security elements were altered or unintentionally reconfigured by the glitch, and thus potentially bypassed/disabled after the restart or reboot.

– *Anomalous events or conditions* are unpredictable, unrepeatable, and sometimes unrecognizable. This is what makes them anomalous! Their direct or indirect effects on software or security elements are determined only in hindsight, or after repeated occurrence, in which case they are no longer anomalous. When software or security system elements are configured to operate in a highly reliable manner, then anomalous conditions are often ignored through use of system breakpoints that are logged and recoverable back to last known point of reliable operation via fast reboot or redundant coprocessors.

– *System outages* denote periods of time when an enterprise software system in unavailable for routine use online (cf. authorized users denied system access for lack of availability). Outages may be scheduled and notified in advance (e.g., for hardware repairs or planned system upgrades), in which case they should not give rise to unintentional vulnerabilities. However, outages may arise unexpectedly for reasons not visible to remote API or system users. Any of the unintentional conditions or events listed here can give rise to unplanned system outages, as may other sources of unreliable system operations. Generally, outages can be mitigated through provision of hot-swap backup or redundant system configurations, but these come at a cost. The ongoing profitability or revenue-positive condition of the enterprise may determine whether or not outages are mitigated through redundant system configurations.

– *System failures* can arise due to the emergence of any of the preceding kinds of unintentional events or conditions that disrupt enterprise operations at an individual, group, or business unit level [Loscocco, Smaller, et al. 1998]. These failures generally require some form of human or organizational intervention, as well as replanning and rescheduling of work in progress, as well as assessing whether or how to recover system managed work products that were unintentionally modified or corrupted as a result of the system failure. Systems failures can trigger consequences like bypassing or resetting system security protections or capabilities back to an earlier version that has already been updated and replaced, thus potentially re-exposing known software vulnerabilities.

– *System implementation failures* denote the failure of an enterprise to completely and properly install and transition to a new software system (or major system version release). Implementation problems are commonly manifest over longer periods of time, sometimes ranging from weeks to months or years. System brought online or into production (or even pre-production) prior to implementation completion may be configured to operate with/without extant enterprise cybersecurity capabilities, policies, or methods in place and operational. Vulnerabilities in such system configurations may allow attackers to covertly enter the system perimeter and to hide/bury itself for an indefinite period until awoken by remote control or system clock. While the inadequately implemented system itself may be vulnerable, it may also simply be exploited as a covert gateway to other enterprise systems of interest to attackers.

Other unintentional socio-technical events, conditions, or acts may be identified, as may their consequences for altering, corrupting, or unknowingly reconfigured cybersecurity system elements, capabilities, or methods.

Overall, it is clear that unintentional socio-technical threats are recurring, inherently difficult to prevent, and entail human-computer activities that are commonly undocumented, not taught, and thus persist. Accordingly, managing unintentional threats will always require vigilant practice by software system users, maintainers, and administrators, as cybersecurity system capabilities and methods cannot in general overcome these limitations.

Countermeasures for Mitigating Cybersecurity Threats

Software systems security mechanisms for implementing security requirements and policies are often employed on an ad hoc basis rather than in a scalable, organized, and effective manner. Convenient, interactive approaches supported by automated evaluation and guidance are not available because there is no formal basis connecting security requirements and policies with the security mechanisms that are to fulfill them. What is available is a palette of disjoint mechanisms or *security countermeasures* for implementing individual system security features [Loscocco, Smalley, Muckelbauer et al. 1998, Spencer, Smalley, et al. 1999] augmented by generalized practices and process standards, such as:

- mandatory access control lists;
- firewalls;
- multi-level security capability lists;
- authentication (certificate authorities, passwords, etc.);
- cryptographic support (e.g. public key certificates);
- encapsulation (including virtualization and hidden rather than public APIs), hardware confinement (memory, storage, port, and external device isolation) [Sun, Wang et al. 1999], and type enforcement capabilities;
- data content or control signal flow logging/auditing;
- honey-pots and traps;
- functionally equivalent but diverse multi-variant software executables [Franz 2010, Salamat, Jackson et al. 2011];

- Security Technical Implementation Guides (STIGs) as user guides for configuring the security parameters for applications [DISA 2011] and operating systems [Smalley 2012];
- secure programming practices (secure coding standards, data type and value range checking, etc.) [Seacord 2008];
- standards for development organization processes and practices rather than system security policies [ISO/IEC 2005];
- anti-virus software that routinely search system repositories for known attack viruses;
- deep data traffic monitoring within/between enterprise databases, websites, portals, or specified client computers that log all data movements, transfers, or updates for secondary analysis, using techniques like machine learning or others;
- standards-based software taggants [Kennedy and Muttik 2011], used by software producers that assert a secure, encrypted identity authentication and provenance to a baseline software element release/version.

The reader will note that these mechanisms are s*oftware 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 integrate and evaluate mechanisms in combination. 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 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 they can be assembled in hierarchical patterns that can be designed and combined in a system architecture to meet specific high-level security policies and requirements.

We believe there is an opportunity to address security requirements challenges throughout a system architecture using *computational security licenses*, licenses whose declared obligations and rights can be formally specified and computationally enforced through automated via executable software programs.

In our previous work [Alspaugh, Asuncion, Scacchi 2009, Alspaugh, Asuncion, Scacchi 2011, Alspaugh and Scacchi 2009, Alspaugh, Scacchi, Asuncion 2010], we showed how software licenses for the components of a system 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 any conflicts and mismatches among licenses in the architecture.

We propose the same approach for security licenses. System architects and analysts can select desired security rights, assign an expected security license to each subsystem or component, and evaluate interactions between these choices at every level from an individual component up to the entire system. Of course assigning a security license to a component does not guarantee that the component's developer will make it satisfy its security obligations, any more than accepting a component under GPL guarantees that the system's stakeholders will satisfy the GPL IP obligations. But assigning a license (whether security or IP) 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, Asuncion, Scacchi 2011], it forms the foundation for design guidance with respect to the issues raised by the licenses, and a means for combining the potentially dissimilar licenses to evaluate their overall interaction and effect, and thus 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.

Security Licenses as Smart Contracts for Specifying Software Cybersecurity Rights, Obligations and Countermeasures

In general terms, a security license is analogous to an ordinary software license such as GPL (GNU General Public License) [FSF 2007]. 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 particularly for 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 expressed in natural language, the rights and obligations are often presented in an intermingled organization, 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.

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

There is no “Securityright Act” analogous to the U.S. Copyright Act [US 2017], or Berne Convention [Berne 1979], to define the exclusive security rights of system stakeholders. 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 Rights within Security Licenses for OA Software System Components

Access Rights

- The right to read data in compartment T.
- The right to add data to compartment T.
- The right to remove data from compartment T.
- The right to delegate security right R.
- The right to read the security license of component C.

Evolutionary Update Rights

- The right to replace component C with another component D.
- The right to update component C to newer version C′.
- The right to revert component C to older version C′.
- The right to add component C in a specified architectural configuration.
- The right to update component C in a specified architectural configuration.
- The right to alter the architectural topology of subcomponent B.
- The right to alter the architecture of system S.
- The right to add security mechanism M in a specified configuration.
- The right to update security mechanism M in a specified configuration.
- The right to remove security mechanism M from a specified configuration.
- The right to replace the security license L of component C with another security license.
- The right to update security license L.

Sample of Security Obligations within Security Licenses for OA Software System Components

Access (Control) Obligations

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

Malicious Software Prevention Obligations

- The obligation for component C to have been vetted by authority A to exercise security right R.
- The obligation for component C to have been vetted by authority A to be the object of security right R.
- The obligation for each component connected to component C to allow it to exercise security right R.
- The obligation for security license L to meet specified criteria.
- The obligation for security license L to be approved by authority A.

Exclusive Security Rights

If there could be legally defined and protected exclusive security rights, what would they be? We nominate the following candidates for discussion:

- The right of the owner of a copy of a system to replace, update, or revert any of its components.
- The right of the owner of a copy of a system to add or remove components or otherwise alter its the architectural topology.
- The right of the owner of a copy of a system to replace or update the security license of the system or any of its components.
- The right of the owner of a copy of a system to alter its user IO streams or ephemeral data. (We envision that persistent data may fall into a different category of protected entity.)

As with the exclusive copyright rights, the owner of a right may license all or part of it to someone else in exchange for obligations, for example to allow a trusted system provider to automatically install certain kinds of updates.

Overall, cybersecurity requirements or capabilities can be expressed in much the same way as IP licenses: using concise, testable formal expressions of obligations and rights. We found that rights and obligations sufficed to express all the software IP licenses that we examined [Alspaugh, Scacchi, and Kawai 2012]. The lists above show example that express security rights and obligations, and in ongoing work (Scacchi and Alspaugh 2017c) we present a model of cybersecurity threats to support a representative set of security issues in OA ecosystems. We envision that during architectural integration security licenses will be created to control how cybersecurity will be supported, as current and future releases of components and applications from external producers are integrated.

OA ecosystems are too complex and fast-changing for a security regime that is not automated to the greatest extent possible. Right-obligation licenses are automatable, and security licenses can made enactable, for example by smart contracts controlling blockchain transactions; in this way, as components evolve and are attempted to be integrated into a new release of the system, the security licenses can require that appropriate obligations are satisfied as an inseparable part of exercising a security right.

In the process of software evolution of the multitudinous parts of the integrated system, security licenses will control which versions are incorporated, under whose authority, and when. This integration can take place as part of the development process or as part of the management of a consumer's installed software configuration. Security licenses give a flexible, computational, extensible, scalable approach to managing ongoing security concerns in a software ecosystem.

Effectiveness, Manageability, Evolvability of Security Licenses

Consider the case of the development of an open-architecture (OA) system integrating proprietary and open-source components from a variety of producers, most of whom do not coordinate their activities and none of whom are controlled by the organization producing the OA system. From the point of view of ensuring security, this is arguably the worst possible case, but it is an increasingly prevalent development model [Alspaugh, Scacchi, Asuncion 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 and Alspaugh 2013] that isolates the component with a hypervisor [Xen 2017] and mediates all communication with the component (method/function calls, data streams, etc.) through shim code that monitors and restricts it.

A typical current-day technique [Luom and Du 2011] for managing security measures is to use capability lists to control each component's access to resources such as function calls and data compartments. 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 and/or forbidden capabilities, managed manually; new capabilities are typically added to the end of the file. As there appears to be no formal model supporting relationships among capabilities, interactions between capabilities are also identified and managed manually. 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. Because a capability list has no hierarchy or recursive structure, managing them is not scalable.

A more sophisticated approach is possible using a declarative policy language such as Ponder [Damianou, Dulay, et al. 2001] or an ontology-based language such as KAoS [Uszok, Bradshaw, Johnson, et al. 2004] that groups capabilities hierarchically, in (KAoS) ontologies or grouped by roles (Ponder). However, they have no provision for organizing capabilities by software components, combined hierarchically into system architectures, and no obvious connection to law.

We contrast the use of security licenses. In some ways, the approaches are similar, in that our candidate security rights are reminiscent of capabilities, and security licenses can also be used to identify and block disallowed operations automatically. However, because many of the actions required for the security obligations are related by subsumption to those granted by the security rights, and many of the obligations are in the context of the component for which corresponding rights are being granted, it is possible to automatically calculate the interaction of rights and obligations throughout the immediate neighborhood of each component, the subsystem containing the component, and so on recursively on up to the system as a whole [Alspaugh, Asuncion, Scacchi 2009].

Structuring cybersecurity rights, obligations and countermeasure (or collectively, cybersecurity policies) as security 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. 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, Asuncion, Scacchi 2011]. Perhaps most importantly, such specification of cybersecurity licenses using smart contracts (or domain-specific languages in which such licenses may be coded) 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.

Blockchains and Smart Contracts for Installed Software Configurations

How might we utilize blockchains and smart contracts to record, track and verify updates to OA software system configurations as they evolve over time while transitioning across software supply chains? We examine this question in this section.

Ledgers of installed software configurations

We envision a new kind of ledger: one that records executable computational updates to the specification of the current installed, operational configuration of C2/B systems of interest. The executable computational updates are similar to scripts in a declarative scripting language, like that used to direct the invocation of utilities on an operating system, procedural scripts involved in building (compiling and integrating) a targeted software executable, or for customizing the functional display and navigation operations within a Web browser. We call the repository in which this specification is recorded, the *installed software configuration* (ISC) ledger. The ISC is the counterpart to a *packaged software configuration* (PSC). The PSC denotes the collection of software elements (e.g., the collection of files and related software execution scripts that will install an integrated mobile app that is ready to use) configured for download and installation on a target software platform or run-time environment. Installing a new PSC into the currently deployed ISC produces an updated and evolved ISC. The ISC specification is therefore a kind of technical data pertaining to the cybersecurity of an OA software system to be managed, tracked/logged, updated, and maintained within an acquisition effort. Such data may be readily managed using a database or other repository capable of organizing and storing update transactions to a software *bill of materials* (BOM 2017), but with the difference that we need a software BOM for both each PSC to be installed, and also the accumulating, evolving ISC.

The ISC ledger records the transactions that update the software applications, including their components, interconnections, interfaces, or licenses for such installed on each machine of interest, such as a desktop PC, smartphone, or central computation server within a mission command or enterprise data center. The installation is enacted via an installation (update) transaction, which may be enabled using an “installation wizard” for a standalone PC application, or a ready-to-install packaged software app acquired from an online app store. For each application installed, the ledger lists the repository from which the software app or update was acquired, the version of the application or update, and some information with which to confirm/verify the version, such as the size of that version of the app, meta-data about where it resides in storage on the machine, other information, or a combination of these. How do we ensure that the repository’s copy is safe, has not been unintentionally modified, and has not been attacked or unknowingly compromised? How do we ensure that attacks are not falsely recorded in the ledger?

In order for a ledger to be up-to-date, each approved installation must be recorded there. How do we ensure this is the case for approved installations? If a ledger is up to date, then an auditor can verify the approved installations by examining the ISC specification for the machine of interest (e.g., a smartphone or laptop PC). Furthermore and most importantly, the blockchain can be queried to identify non-approved or non-compliant installations, whether these are apps or updates that were innocently installed but not recorded in the ledger, or maliciously injected software for some nefarious purpose, and thus such covert updates are not recorded in the ledger. In either case, the auditor can then institute for each application that does not match the ledger a rollback to a known safe ISC state matching what has perviously been verified on the ledger.

The following issues must be managed appropriately for the ledger scheme to succeed.

- ***How is it ensured that the origination or destination repository’s copy is safe and has not been attacked?*** This is a separate concern, and one that is equally problematic with or without a ledger system. We do not discuss it further here, merely noting that it must be ensured for devices to remain secure. But in normal operation, the ISC specification has a unique identifier, denoted by the *hash code*[1] value associated with the current system when last updated and subject to remote verification by anonymous miners who may be unknown to the system integrators. This hash code may reveal whether the ISC specification copy's hash code matches the one checked during audit or subsequent miner verification activities. If the hash code values are different, then something has altered the copy, and thus it may be rolled back to a prior verified state or ISC specification.

- ***How is it ensured that every approved installation or update is recorded in the ledger?*** The ledger system must be integrated with whatever system manages installations and updates for the machines in question. We note that unapproved installations or updates can be automatically detected and can be rolled back or reverted at the next audit point/event, so there will be a strong motivation to ensure that desired transactions are recorded.
- ***How do we ensure that attacks are not falsely recorded in the ledger?*** Obviously this is a key concern. As discussed below under Transactions, changes to the ledger are validated by multiple autonomous parties (miners) using several sources of information, and each particular copy of a ledger competes with all others for accuracy as part of the blockchain scheme.

Transactions for installed software configurations

Each transaction in a ledger records an installation or update of an app on a specific machine. How do we ensure that all valid installations or updates are presented? Every time a new application is installed, or an existing application is updated, the appropriate information is recorded in the ledger. If an application is installed or updated without being recorded in the ledger, that installation or update is recognized as unverified, and thus rolled back the next time the machine is audited. Audits may simply involve checking a hash code value (a long, non-guessable string of characters that is computationally generated within the blockchain system), or a similarly unique software taggant hash code [Kennedy and Mutitk 2011] associated with the current ISC specification on the target machine, with the corresponding value in the blockchain--this is a simple match-checking query that can be performed periodically, or by enterprise policy. When the audit reveals a mis-match, then a roll-back may be triggered that reverts the ISC on the

[1] A "hash code" is the result of a computation that invokes any hash function that can be used to map data of arbitrary size to data of fixed size. The values returned by a hash function are called hash values, hash codes, digests, or simply hashes. A cryptographic hash function allows one to easily verify that some input data maps to a given hash value, but if the input data is unknown, it is deliberately difficult to reconstruct it (or equivalent alternatives) by knowing the stored hash value. (cf. *Hash Function*, Wikipedia 2017).

machine to a previously trusted ISC, and then remove, deprecate, or flag the unverified ISC as suspect, along with distribution of notification to relevant parties of such action following enterprise policy. But how do we ensure that only valid installations or updates are presented? Transactions that would record an invalid installation or update, fraudulently misrepresenting the repository's version's size or hash or from an untrusted repository, are identified by comparison with the set of trusted repositories, with the size and hash information recorded there for the installation or update in question and for the data calculated from the destination machine afterwards. Accordingly, we are acting to use blockchain techniques as intended, but for a new kind of use case, namely that of ISC specification update, verification and reconciliation.

Smart Contracts for installed software configurations

A smart contract works within the framework of the blockchain ledger and transaction system, ensuring that the required obligations for each transaction are met before the transaction is enacted, verified, and then recorded in the ledger. These obligations are associated with those we have previously identified and specified as security requirements for insuring access and update rights encoded in a software system's security license [Alspaugh and Scacchi 2012].

An example ledger, transaction, smart contract implementation system

Ethereum [2017] is being used used to implement smart contracts, transactions, and a blockchain ledger. Ethereum is a set of technologies: a general-purpose programming language, open application program interfaces (APIs), and an open transaction/blockchain repository associated with the APIs. Ethereum uses a cryptocurrency called *ether*, and users of Ethereum can transfer money, ownership, or control of exchanged resources whose (fungible) value is denominated in the form of ether between each other and to contracts to hold in escrow. Online currency exchange markets can exist for converting ether to a traditional currency like US dollars. Users of Ethereum send transactions to it in order to create contracts, invoke existing contracts, and transfer ether. The transactions are public and permanently recorded in the blockchain, unless access to the blockchain is restricted/private to an authorized set of known parties who must be granted permission to access or update the blockchain.

Ethereum is decentralized, with a network of blockchains for which each transaction is processed by a number of *miners*, possibly anonymous actors who perform computations on the blockchain that collectively verify the validity of a transaction of data/value between the participating parties. These miners are mutually-untrusted peers who are paid fees (in ether) for the work of processing each transaction and its contract provisions. A miner groups transactions into blocks and performs a calculation (or “solves a puzzle”) that takes as inputs the previous block in the blockchain and the transactions in the new block. A *valid* block, one whose puzzle has been solved and which meets certain other conditions, can be appended to the blockchain. The miner broadcasts the new valid block to the network and receives the ether paid for each of the transactions by their originators. In this way, Ethereum-based smart contracts are validated by decentralized miners. These miners receive payment when contracted transactions they verify are successfully appended by consensus to the blockchain.

A transaction may appear in a number of different blocks, produced by different miners and appended to different blockchains. Ethereum pays miners somewhat more to append a block to a longer blockchain, which has the effect over time of converging the ledger to the blocks and thus transactions that the majority of miners agree are valid.

THIS PAGE INTENTIONALLY LEFT BLANK

Blockchains and Smart Contracts for Managing Software Development and Evolution Process Transactions

How might we utilize blockchains and smart contracts to manage software development or evolution updates to OA software system configurations over time? We examine this question in this section.

Continuous Software Development and Evolution Processes for Open Architecture Software Systems

In previous work, we have identified and substantiated seven types of software evolution process update transactions, shown in Figure 7 below. We further observe that a given software evolution process may entail either (a) one type of transaction per update, or (b) multiple concurrent types of updates per transaction. This may be due to current-to-evolved transformations where the evolved system version of the OA configuration involves the replacement of more than one component arising from the availability of a new technology that represents a departure from the current system architecture, or that integrates functionally similar capabilities through a new mix of components, interfaces and interconnections (e.g., when combining multiple widgets into mashups [Endres-Niggemeyer 2013]). The purpose may be to reduce software maintenance complexity and extend the sustainability of a deployed current (or legacy) system through adoption and integration of remote (cloud-based) services that are functionally similar to the capabilities formerly available in multiple components. For example, replacing legacy office productivity applications (word processor, email, calendar) with browser-based remote networked services (*Google Docs*, *Microsoft Office 365*), can provide end-users with functionally-similar processing capabilities, but with fewer application components installed on the end-user's desktop PC system. Furthermore, subsequent updates to remote services may by policy be integrated and deployed automatically for minor functionally equivalent evolutionary updates (e.g., bug fixes), or by deployed only by request or authorization when functionally similar system version updates are made available [Scacchi and Alspaugh 2013a, Scacchi and Alspaugh 2015, Scacchi and Alspaugh 2016, Scacchi and Alspaugh 2017].

Ledger: what versions of what software components and connectors are integrated in what OA configuration topology

A ledger records and defines through the design-time OA specification, the ecosystem in which the OA is evolving [Scacchi and Alspaugh 2012]. The OA is represented using an architecture description language, and successive ledger entries record successive configurations of the OA system as it evolves. The ledger as a whole presents the history of the OA's evolution, and as long as the components and connectors remain available from their repositories an instance of any stage of the OA can be rebuilt as needed. At a minimum the ledger records every release of the OA system.

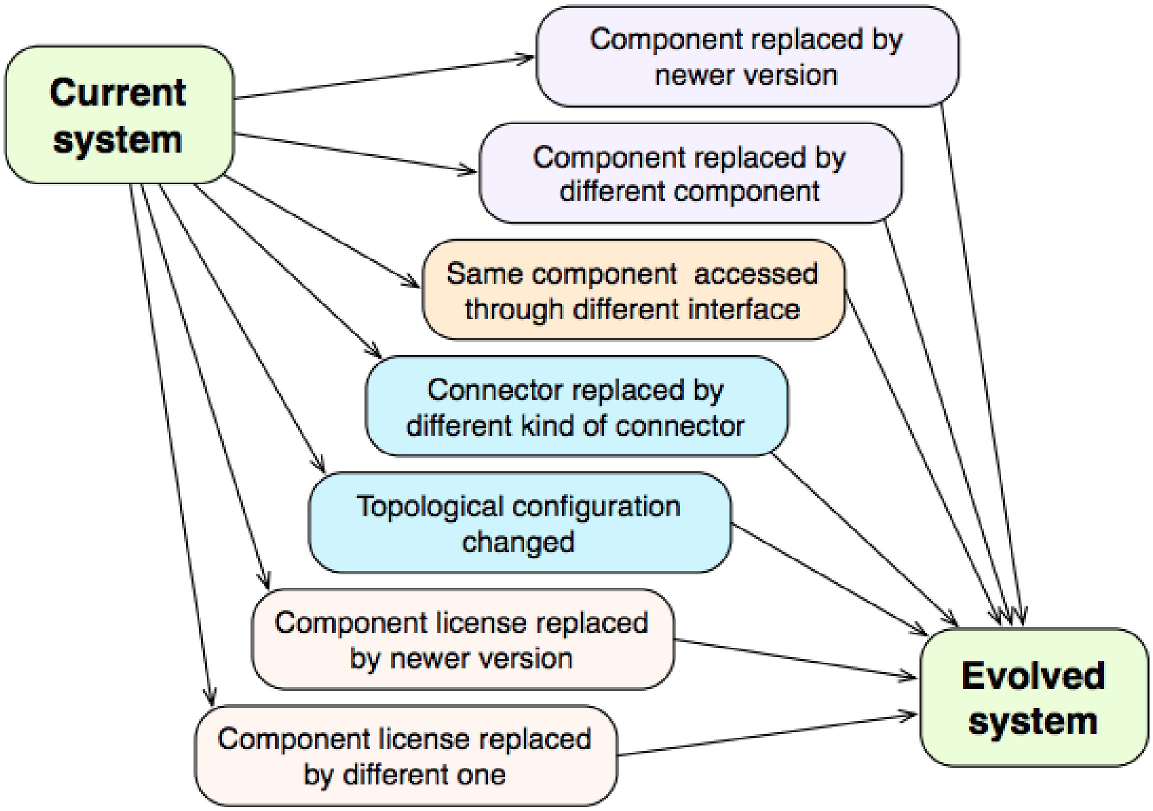

Current system
Component replaced by newer version
Component replaced by different component
Same component accessed through different interface
Connector replaced by different kind of connector
Topological configuration changed
Component license replaced by newer version
Component license replaced by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

Figure 7: Seven types of software evolution update transactions [Scacchi and Alspaugh 2012, Scacchi and Alspaugh 2017a].

If a machine on which the OA ISC is installed needs to be rolled back to an earlier configuration, the desired version of the ISC can be rebuilt guided by the corresponding ledger entry.

Transactions: OA evolution steps

Each transaction corresponds to one (or several) of the seven types of OA evolution, stating which component, connector, or license is being changed or what change is being made to the OA topology. In total, the sequence of all transactions for an OA system represents the history of its evolution. The ledger summarizes the system's evolution, based on the transactions made to it, and presents each of the versions that the evolution has proceeded through.

Not everyone can record a transaction with the ledger, and each actor that can record a transaction may be restricted in precisely what sorts of transactions can be recorded. These restrictions ensure that the OA ISC is evolved through steps that preserve its security. It also accommodates actors who may or not have been vetted and authorized, so that they are trusted to preserve the system's security through their transactions.

Smart Contracts: enforcing obligations for each OA evolution step

Smart contracts restrict the transactions that may occur to those believed to preserve the OA system's security as the system evolves. A transaction may only be enacted if the actor doing so has been vetted and authorized for it, and has presented credentials identifying himself appropriately; and also only if the current state of the OA system development and the evolution step(s) proposed meet the conditions imposed by a smart contract associated with the ledger. The smart contract in essence states obligations that the actor, the evolution step, and the OA system must meet in order for the transaction to occur; if the obligations are not met, then the transaction cannot be performed, at least not with this smart contract. The obligations declared in a smart contract indicate which parties or actors can access/update what OA system elements or other technical data arising during software development or evolution processes. As before, these process obligations are similar to those previously identified for controlling software system/data usage obligations, along the rights to access and update the system/data provided to developer, system integrators, or end-users [Alspaugh and Scacchi 2012].

It is possible that more than one smart contract may potentially allow a specific transaction, each contract presenting a different set of obligations. But in any case the transaction cannot proceed until a smart contract for the ledger allows it to do so.

To help make clear what we are looking to accomplish through our efforts to stimulate innovation in securing the development and evolution of OA software systems, we now turn to present a case study focusing on updating the installed software configuration of a deployed current OA C2/B software system.

Case Study: OA C2/B Software System Evolution Process Updates

In this case study, we describe how blockchains and smart contracts can be employed to model and analyze cybersecurity requirements for OA software systems that arise during software evolution processes. As described above, there are seven types of software evolution process updates that take a current system, transform it one of the seven ways, which produces an evolved system. This evolution process iteratively cycles through software development processes that build, release, and deploy [Scacchi and Alspaugh 2013b, Scacchi and Alspaugh 2017] installed software configurations once the development life cycle starts. The process continues to (slowly) cycle over time, until the system is retired or abandoned. Our focus further narrows to evolving OA C2/B systems that incorporate multiple end-user computing platforms, such as smartphones, tablets, or other Web-compatible "edge" devices [Zheng and Carter 2015], as we have addressed before [Scacchi and Alspaugh 2015, Scacchi and Alspaugh 2016].

Blockchain ledgers serve to verify in a decentralized manner the proper sequencing of valid transactions to user/device account. Such an account operates like a personal bank account that can be used to deposit and withdraw funds, for example, through use of account transactions associated with debit/credit card bound to the account. The enterprise that manages accounts for users may charge a fee for account transactions, though such fees may be assigned to a third-party (e.g., party who receives a payment via a card that has been authorized to possess sufficient funds balance to cover the payment in the future). The current "balance of funds" in a software evolution process account indicates the name, size, and other meta-data that identify executable software applications (including mobile apps, plug-in widgets, or other installed software). At present, computing platforms or devices do not maintain software process transaction accounts, but in our scheme they would.

Next, the blockchain ledger as a decentralized database would be distributed across a (virtual private) network of computing systems, such as those with restricted, authenticated access to a centralized C2/B system host/sub-network. Said differently, if we have smartphones or mobile/laptop PCs that can roam in the wild, and intentionally or unintentionally acquire software updates (e.g., known app updates but with revised access rights; new social media apps; or cyber-penetration attack vectors via misdirected access to a remote server), we want all such evolutionary software update transactions to be reconciled and validated against a the corresponding virtual private network's blockchain ledger in ways that maintain device/user autonomy, but reveals and can reject unvalidated evolutionary updates. The ways and means for how valid or invalid transactions are revealed (externally documented on the blockchain) or rejected (e.g., enforced automated uninstallation, external network access blocked, or notify user of problematic update) are determined by enterprise cybersecurity policies encoded into an associated smart contract (a functional software program logically isolated from end-user application software).

Let us consider the following usage scenario. Suppose we have a mission platform like a ground-based command post (or remote enterprise business office) assigned to operate within an international location. Such a location may be in a region known to have a history of prior cybersecurity attacks on personal computers, mobile, or Web-based devices that access the public Internet. Mission personnel are restricted by policy from using their enterprise mobile devices outside the cybersecurity perimeter of the mission platform. However, personnel may also possess and use private personal devices, such as low-cost smartphones that are used for non-mission purposes.

As anyone who possesses and routinely uses a mobile/edge device like a smartphone or laptop PC now frequently experiences, software (evolution) updates are common, sometimes one or more per week across the 30-60+ apps found on such devices. Sometimes mistakes are made by personnel regarding which device to use for accessing remote services like making phone calls to home, to informally coordinate with friends in allied forces, to check for local restaurants offering interesting local cuisines, or to post data for sharing on social media. Access control to some devices may be misconfigured due to a prior update or unintentionally left open in an discoverable device pairing mode, so that other unknown devices or remote computers can quietly/covertly make network connections that enable data/files upload, download or remote control. Mobile or web-based edge devices will be relentlessly targeted for cyber attack, so when a cyber attack vulnerability is in the hands of opposing forces or hostile competitors, we assume they will seek out and attack these vulnerabilities at some time and place. It is therefore these invalid software evolution updates to installed software configurations that denote potential cyber attacks that we seek to detect, isolate, trace, expunge or prevent, using the capabilities of blockchains and smart contracts. In this way, our use of blockchains and smart contracts is innovative, original, and not previously associated with software evolution process transactions.

Consider a desktop PC with apps/widgets acquired from either a restricted-access enterprise-specific app store, a Defense app store [George, Galdorisi, Morris, O'Neil 2014, George, Morris, O'Neil 2014], or else from a public-access app store or OSS component repository. A sample picture of such an ISC appears in Figure 8. This ISC includes web browser-based apps like cloud-based word processors, calendars, and email app services are frequently included in such stores. However, open access app stores like those operated by Apple, Google, Microsoft and others also offer free/low-cost apps that offer many other remote, cloud-based services. In either situation, these remote service apps may operate downloaded software code that runs within a platform-based Web browser that accesses public or (virtual) private networks. Enterprise end-users with computer programming expertise may even create and integrate multiple apps/widgets into mashups as a kind of end-user software evolution process update [Endres-Niggemeyer 2013, Scacchi and Alspaugh 2015]. These mashups may enable the participating apps/widgets to interoperate, exchange or update local data, or transfer data/files to/from remote networked repositories [Scacchi and Alspaugh 2015, Scacchi and Alspaugh 2016].

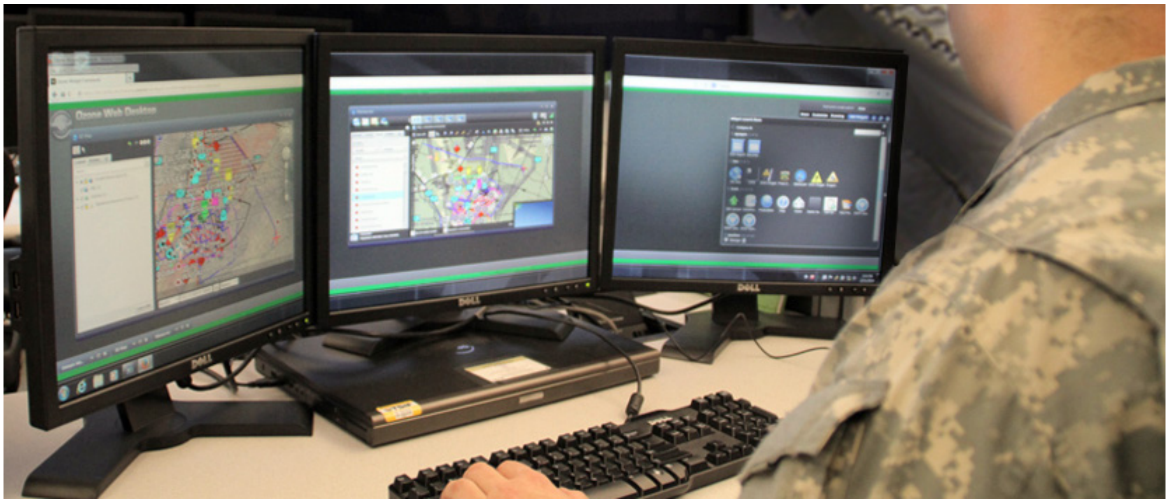


Figure 8. A sample view of a desktop PC within a C2/B installed software configuration supporting multiple OA software system components or apps from an online store.

Next, the desktop PC system may itself by part of a larger integrated OA C2/B system configured to operate within a local area network, connected to a wide-area network supporting remote communications to other command or field operations centers. An example of a such system integration is shown in Figure 9.

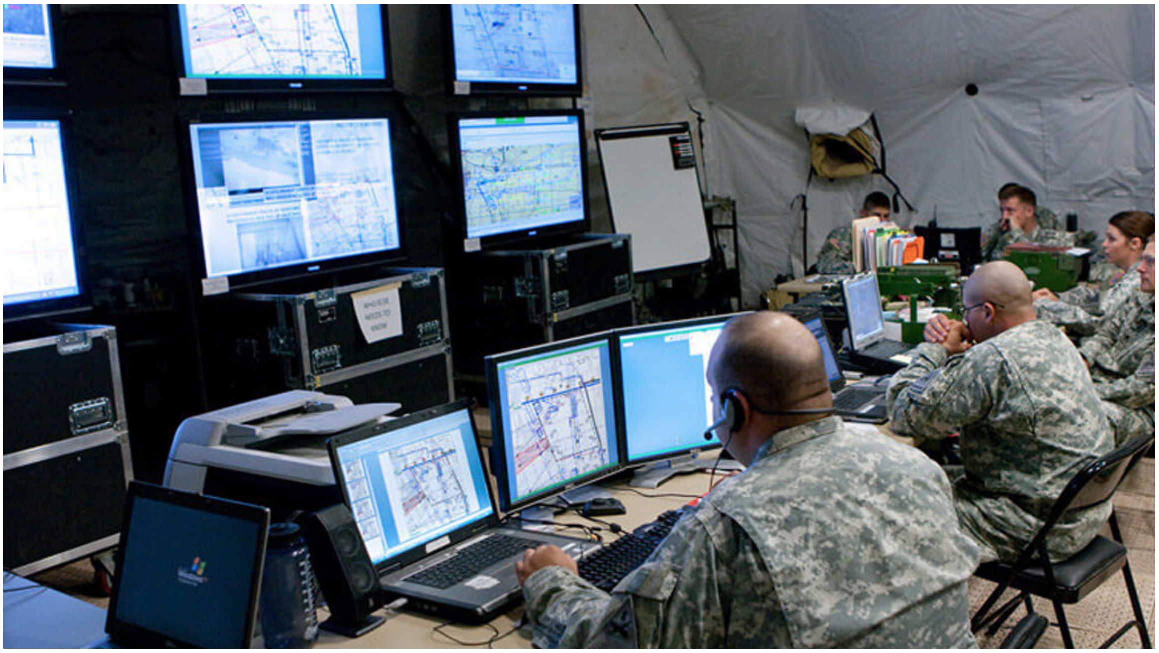
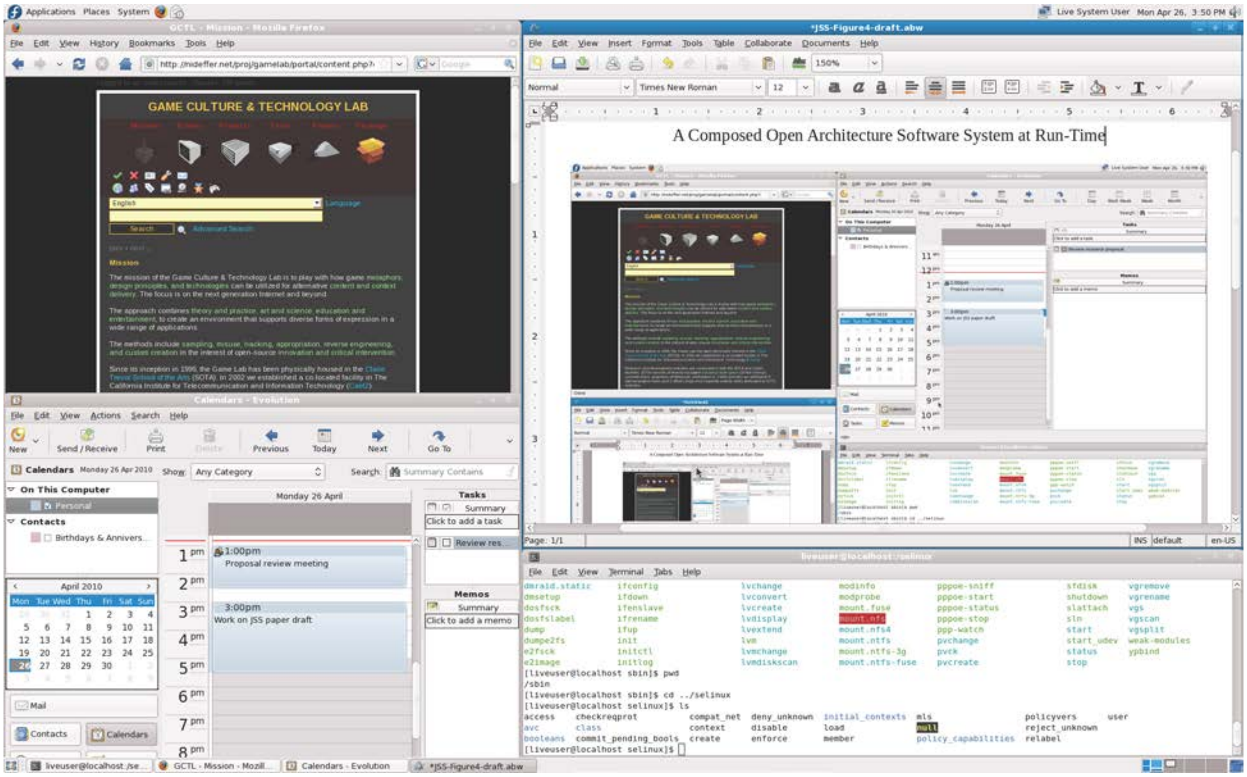


Figure 9. A view of an integrated OA C2/B system configured to operate as a Future Command Center.

If our mobile device is a laptop PC, its current (or legacy) OA software configuration may include open source software (OSS) or proprietary closed source software (CSS) versions of a common Web browser, word processor, email, calendar, and more hosted on the PC's operating system. For instance, a laptop may have a *Firefox* web browser (OSS), *AbiWord* (OSS) or *Microsoft Word* (CSS) word processor, *Gnome Evolution* (fOSS) or *Outlook* (CSS) for email and calendaring, and host PC operating like a *Fedora/Linux* distribution (OSS), *Microsoft Windows* (CSS), or *Apple OSX* (CSS and OSS). The deployed, run-time executable version of this OA ISC system on the laptop PC may appear to an end-user as an array of loosely-coupled applications, such as displayed in Figure 10 below. Now, suppose a decision has been made to update this OA ISC system, to evolve it from the current configuration to one where the word processor, email and calendaring applications hosted on the laptop PC are to be replaced with functionally similarly remote Web services that will operate within the existing Web browser. These remote services thus entail reliance and usage of browser-based software components that are hosted in the cloud and downloaded on user demand. This transition can simplify and reduce the costs of corresponding software update services associated with locally hosted applications (e.g., recurring license fees for CSS elements). The resulting deployed and evolved laptop PC software system may appear to the end-user as shown in Figure 13 below.

Each type of software evolution process update can have a smart contract associated with it. Each such contract programmatically specifies what computational actions need to be performed to complete the transaction with the affected technical data and associated data repositories, and similarly what actions need to be performed on the blockchain. Let us consider the following transformation of a current ISC shown in Figure 10 to an evolved ISC seen in Figure 13. Figure 10 corresponds to its ISC model visualized in Figure 11, which is derived from its specification in an architectural description language (ADL), as we have established before [Alspaugh, Asuncion, Scacchi 2013a, Alspaugh, Scacchi, Asuncion 2010]. As the current system, we assume for this moment, that it has previously been submitted via an earlier transaction on the blockchain that was verified by miners and thus is now a recorded part of the blockchain. Thus we can determine the provenance of the current ISC system and its specification. This blockchain contains a record of the ISC specification and the results (e.g., blockchain hash code values) that the miners computed and agreed by anonymous vote to denote the ISC installed and operational on the target machine/platform. The transformation from this current system to the evolved system thus entails enaction of the associated smart contracts associated with a set of embedded evolution update transactions that collectively denote what updates must be verified as a block for the evolved ISC specification to be appended to the blockchain.


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
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
April 2010
Mon Tue Wed Thu Fri Sat Sun
Mail
Contacts
Calendars
*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
liveuser@localhost:/selinux
File Edit View Terminal Tabs Help
liveuser@localhost:/se...
GCTL - Mission - Mozill...
Calendars - Evolution
*JSS-Figure4-draft.abw

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 1996, 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).

1 pm 1:00pm Proposal review meeting

3 pm 3:00pm Work on JSS paper draft

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 10: Current deployed OA C2/B ISC corresponding to Figure 11, utilized by end-users: *Firefox* Web browser (upper left), *Evolution* calendar (lower-left), *AbiWord* word processor (upper right), *Fedora/Linux* desktop operating system platform (lower right).

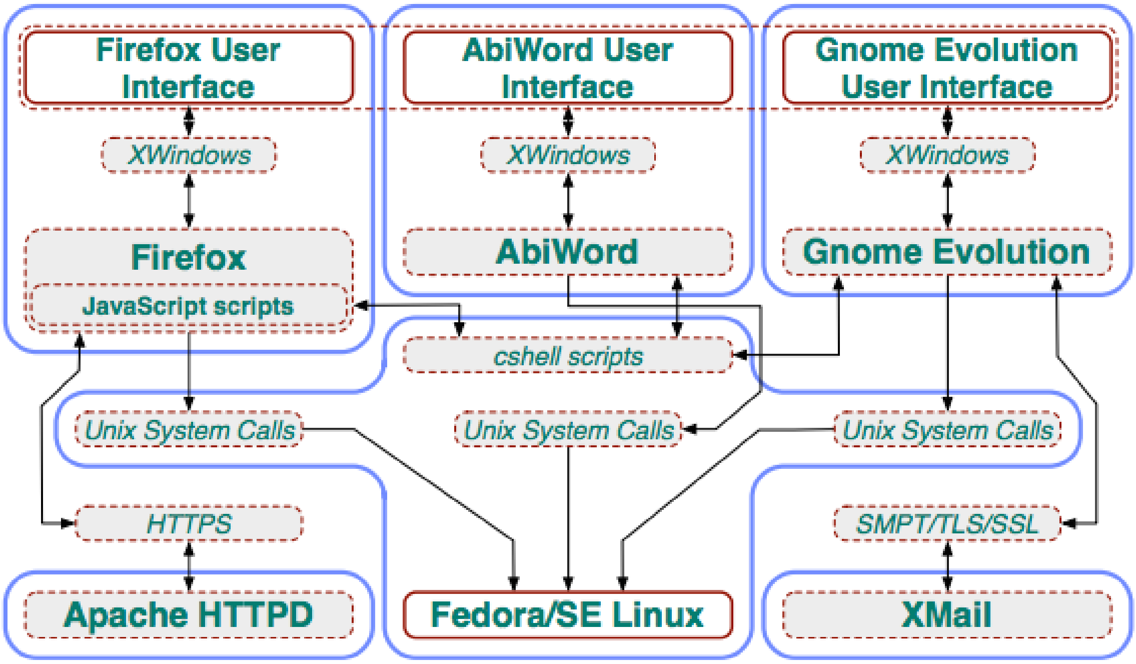


Figure 11: The current ISC specification for an OA C2/B system within security containers at *build-time* [Scacchi and Alspaugh 2013b], intended to denote a record on the blockchain for which components need to be included during integration (and testing) of the software components and code APIs within the released and deployed ISC.

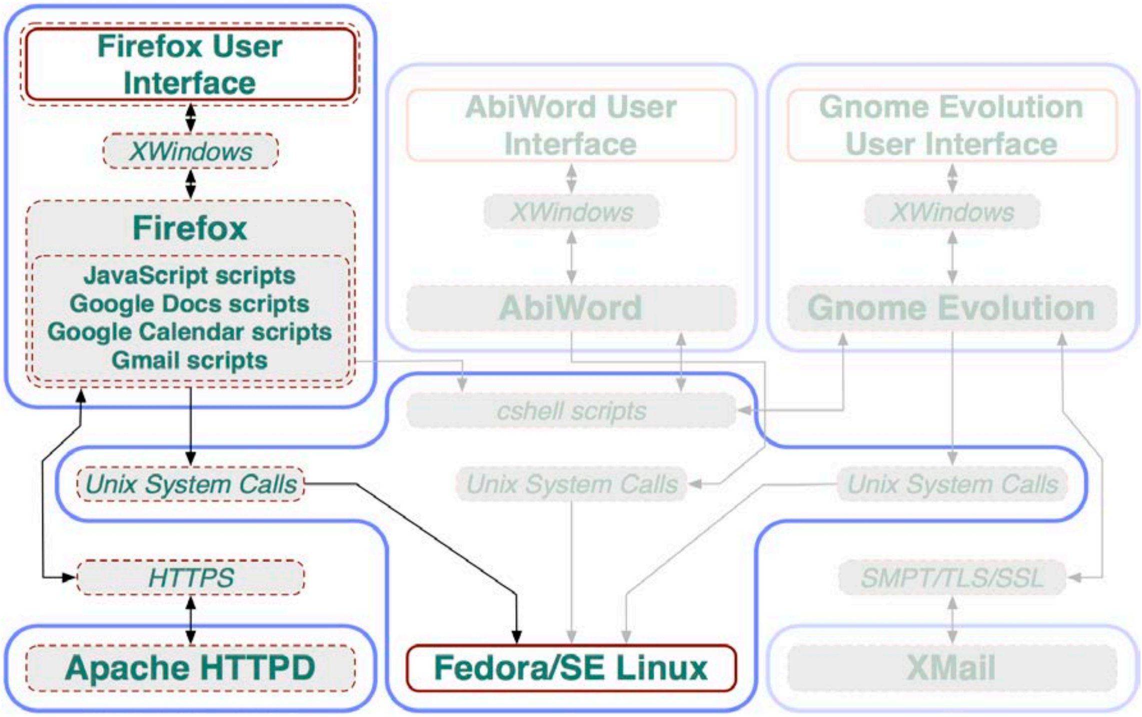
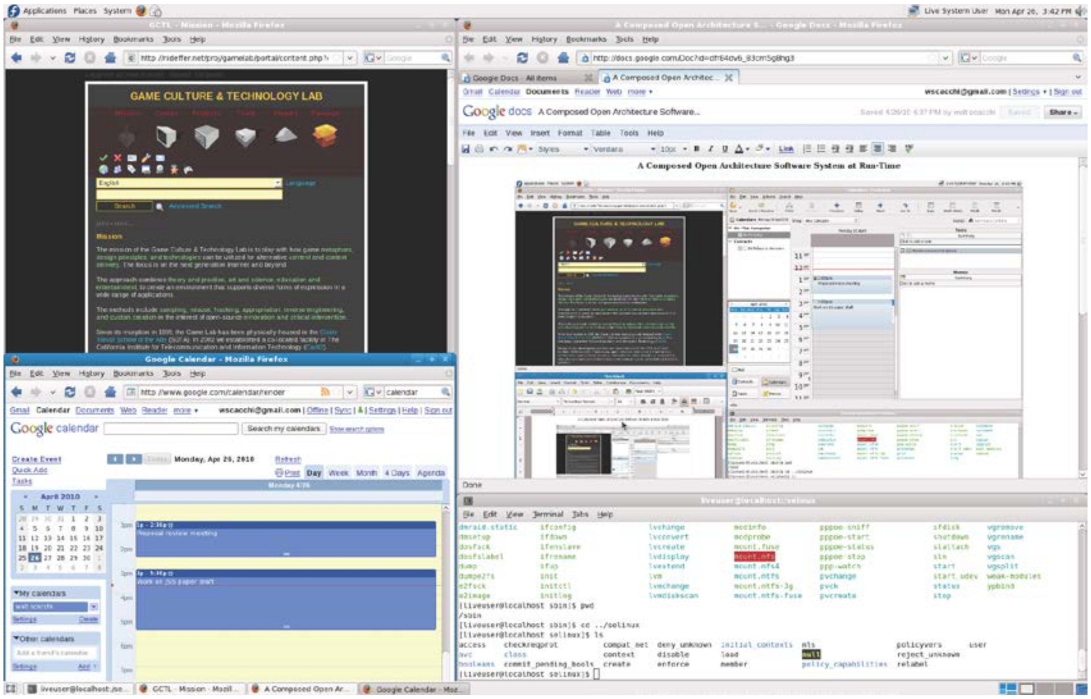


Figure 12: The evolved OA C2/B ISC specification at build-time. Note how the topology of the ISC has evolved, including where now legacy components have been deprecated.

For example, we may elect to use a pre-defined smart contract (an executable software script) whose transactions transform a component-based C2/B system with a Web browser installed, into a remote service-based C2/B system, where Web/cloud-based services provide functionally similar capabilities to end-users. This might entail a smart contract that performs the following transactions (described in English for simplicity): (1) check the ISC blockchain hash code value(s) match those for the current system, if matching, then proceed; (2) deprecate and replace designated software application components with remote service apps/widgets; (3) replace deprecated component licenses with remote services licenses (e.g, ToS); and (4) replace ISC interconnection topology with the evolved ISC; (5) send request to miners to independently compute and verify the evolved ISC specification hash code value on the target machine/platform denotes the ISC and associated meta-data they independently build to compute the evolved ISC hash code; (6) if miners vote independently verifies the ISC specification, then assert into the blockchain the evolved ISC specification value as denoting the new current ISC ready for use; (e) end of contract transactions. Many low-level details are not described here, but would need to be in a smart contract. These details can include, for instance, the installation parameter settings that are selected or configured by either the end-user or installation script, in line with a security technical implementation guide (STIG) for the targeted machine/platform.

The software evolution conveyed in the smart contract example will change the topological configuration of software components found in the system integration build specification, release, and deployed run-time architectures. Here we see that in Figure 12, the configuration model of the evolved OA system still incorporates the same kind of components as the current system model (shown above in Figure 11), but now the topology of components interconnections and interfaces has been updated to realize the deployed, run-time desktop software. Last, a transformation from the current software components with their respective licenses, to the evolved configuration will also entail an update to new licenses (e.g., *Google Terms of Service*), and how these components will be secured (from end-user level assurance of locally installed components to end-user agreement with remotely provided component security that is mostly invisible to end-users).



```
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 13: Evolved OA C2/B ISC corresponding to Figure 12, installed for utilization by end-users: *Firefox* Web browser as before, *Google Calendar* (lower left), *Google Docs* (upper right), and *Fedora/Linux* operating system platform as before.

The transformation of the current system in Figure 10 and Figure 11 to the evolved system in Figure 12 and Figure 13 entails multiple types of software system evolution updates. But now we must consider whether and how such evolution process transactions potentially allow for introduction of cybersecurity vulnerabilities or attack vectors. This can happen, for instance, in the following ways. If the current system is trusted, because its components have individually had their security tested for known vulnerabilities and have passed assurance checks, then evolution process update transactions may introduce unintended vulnerabilities, either within the components replaced, within the new topological configuration, via shifts in the obligations or rights (added, subtracted, revised) in the new components, or via the overall incorporation of all of these evolutionary updates. So we need to assure the security of the update transactions acquired from the component producers and from the system integrators. This entails identifying and validating the software supply network that provides the software components that are included in a new PSC for installation, or as currently configured within a deployed ISC, as suggested in Figure 14. Similarly, when a planned and authorized PSC is to be installed into the evolving ISC, its software supply network that supplies the new PSC for installation and evolutionary update of the current ISC, then its network must also be recognized and validated as the source for the updated OA software system components. A similar example appears in Figure 15.

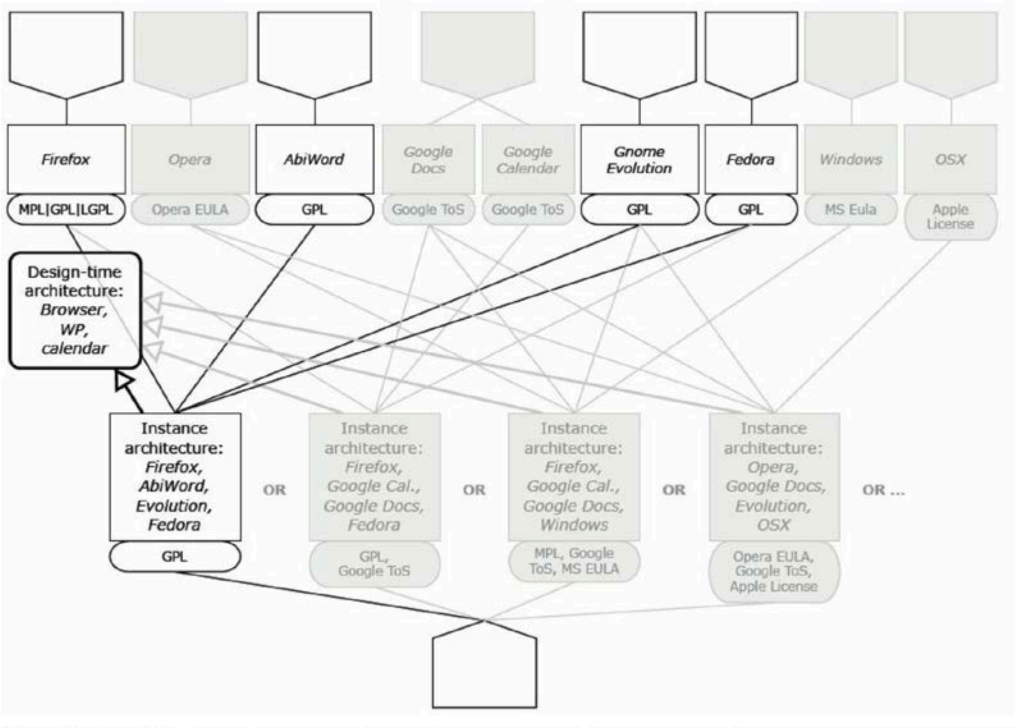

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
GPL
OR
Instance architecture: 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 14. A software supply network for the ISC in Figure 10 and Figure 11.

As these transactions entail request-response transactions with remote parties across a network, then they may be vulnerable to “man-in-the-middle” attacks, as well as to mistakes made in selecting the appropriate component versions for the specific edge device platform. So we want these transactions to be coordinated and tracked using blockchains and smart contracts, so that we can better trust the security of the evolution process updates. Said differently, we want any and all updates that affect the OA software system components, interconnections and interfaces, or licenses to be mediated and verified by remote parties via blockchain transactions. This entails that each edge device or system platform must be able to periodically (e.g., daily, after an application program exits, or by mission-specific policy) identify itself and assert the “value” of its current ISC elements and configuration specification, in a way that can be reconciled against the last known, corresponding verified values on the blockchain. If a discrepancy between the value of the last known (and trusted) current system configuration, and the system evolved configuration is detected, then some unknown evolution update has occurred, such that system security is now unknown and may no longer be trusted. Such a condition may then produce a notification of such discrepancy, automatically revert to the last known trusted current system, or some other intervention action, depending on the evolution process update security policies expressed in the corresponding smart contract. Subsequently, we now have new ways and means for assuring, detecting, or preventing authorized/unauthorized evolutionary changes to an OA ISC during the software development and evolution processes which occur routinely during a system acquisition effort.

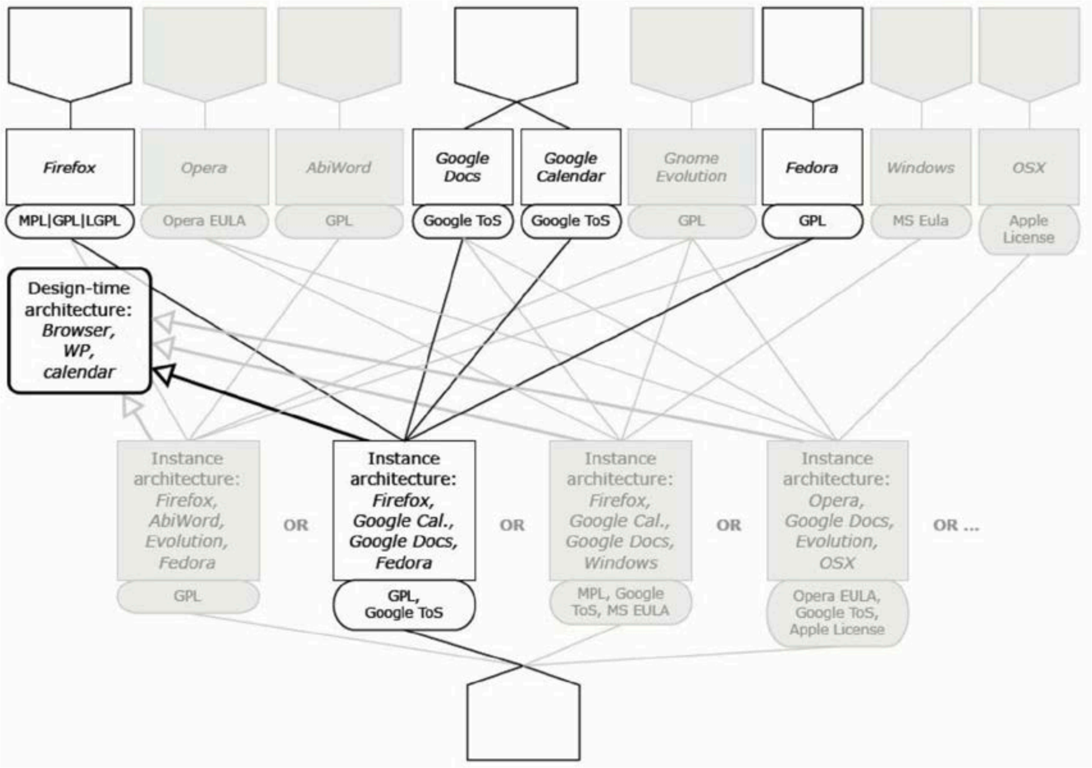

Firefox
Opera
AbiWord
Google Docs
Google Calendar
Gnome Evolution
Fedora
Windows
OSX
MPL|GPL|LGPL
Opera EULA
GPL
Google ToS
Google ToS
GPL
GPL
MS Eula
Apple License
Design-time architecture: Browser, WP, calendar
Instance architecture: Firefox, AbiWord, Evolution, Fedora
GPL
OR
Instance architecture: 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 15. The alternative software supply chain given rise to the evolved ISC in Figure 12 and Figure 13.

Overall, the purpose of this case study is to help describe and reveal that common and widespread acquisition processes associated with the development, usage, or evolution of OA software systems supporting C2/B mission applications is not necessarily secure, and thus can allow for unknown or poorly understood evolutionary updates that are intended or not. Our efforts begin to characterize the need to continuously secure and assure these software engineering process updates and their provenance. Such continuous assurance capabilities are needed in addition to other techniques that focus on assuring the security and integrity of the individual software components acquired from diverse producers or integrators through software ecosystems that release deployable run-time software applications or remote services.

Discussion

There are three topics we find merit consideration, given what now appears possible in the use of blockchains and smart contracts as mechanisms for assuring software development and evolution process update transactions for OA C2/B systems. These are (a) how cyberattacks that may potentially arise in traditional software engineering processes can now be prevented, detected or marked for action; (b) innovations in acquisition research that may follow; and (c) future extensions of this line of research and study.

Cyberattacks on software evolution, release, and update processes

The types of software evolution updates in Figure 7 also classify comparable types of software supply chain threats/attacks on OA systems during software system development, build, deployment, and run-time processes [Scacchi and Alspaugh 2013a, Scacchi and Alspaugh 2013b, Scacchi and Alspaugh 2013c, Scacchi and Alspaugh 2017a, Scacchi and Alspaugh 2017b]. The difference being that cyberattacks on software denote unauthorized or unverified updates from the current ISC during design-time, build-time and deployment-time software engineering activities, to an evolved ISC. This implies that covert software evolution changes by an attacker may follow the same steps as those by a trusted software producer or system integrator; namely replacement of a component by a newer version or by a different component, access to a component through a different interface, replacement of a connector, or replacement of the topological configuration. (We are presently unaware of attacks involving replacement of a component license, but such attacks that change/rewrite IP or security license obligations and rights [Scacchi and Alspaugh 2012, Scacchi and Alspaugh 2015, Scacchi and Alspaugh 2016] are clearly possible.) The result is a compromised version of the system that is functionally similar to the current (trusted) ISC system, but masquerading as one that is authorized, validated, and functionally equivalent intended not to be recognized as something different.

When the attack is made on a deployed instance of the ISC system, its presence can be identified by the change in the size or hash code value of the compromised system, compared to the current system's provenance or software taggant values already established and validated in the blockchain. The window of time during which the attacked system may take effect is limited by the frequency with which the edge device's software is compared with what the blockchain ledger recorded as being installed, as after any change is discovered the edge system's software can be rolled back to its (prior, now current) trusted configuration.

The process is more complex for intentional but covert attacks during development, build, and deployment, because the context is more complex, as indicated in our examination of recent social and technical threats/attacks software supply chain identified earlier in this report. Specifically, we wish to prevent insecure components, connectors, and configurations from being incorporated into the OA system; but an OA system is by its nature typically the result of a distributed, decentralized development, with components coming from other projects and developed and evolved by parties distant and often unknown to the OA system's integrators. We foresee the use of blockchains for PSC/ISC update transactions that are subject to smart contracts within DAO software supply networks to record each component and connector's provenance, vetting, and authorization. Smart contracts restrict the possible transactions (evolution steps) to those believed to preserve the OA system's security. When an unexpected change is discovered in an edge device system's software, it is rolled back to a safe version; when a security fault is discovered in a version of the system, a process that may be much more involved, the components, connectors, and topology involved may be rolled back to a trusted safe version, and the smart contracts through which the fault was introduced may be updated to prevent a "similar" evolution in the future. This may be done either by withdrawing authorization from actors involved, by blacklisting a component repository whose vetting was careless, or by similar means. The blockchain ledger records the information needed to take such steps.

This points to two further areas of research. First, the blockchain ledger system now becomes a locus against which attackers will wish to operate, and further study is needed to examine how to resist such attacks, isolate their effect, and to the extent possible reject them through the blockchain and transaction mechanism itself. Second, can the ledger be used as a database of information for effectively distinguishing fraudulent or corrupted evolution steps? Further research will be necessary.

The only allowed OA evolution updates of the secure system are those that are first verified as valid updates, from known trusted parties, and that satisfy a contract for the blockchain ledger. In cases where a vulnerable or corrupted component, connector, or topology successfully runs this gauntlet, the ledger provides a means for rolling back transactions to a secure version of the system that can be deployed in place of the insecure later version.

We note that in contrast to a procedural programming language such as the Solidity language used for Ethereum contracts, a declarative scripting language mitigates against recently discovered vulnerabilities of smart contract technologies such as those found for the Ethereum run-time interpreter [Atzei, Bartoletti, Cimoli 2016].

Innovation for Acquisition Research

The work prior to this paper in software cybersecurity is primarily focused on making a particular version of the software system itself, as a product, secure. In this paper, we are expanding our view to include the ecosystem within which the system evolves, the software architecture specification that defines and constrains that ecosystem, the evolution of the components and connectors that are integrated into the system, and the OA evolution process by which any OA system evolves from version to version. To this, we are adding the ability to record, track, verify, and maintain the security of the OA system throughout its development and evolution processes.

We are proposing the use of blockchains and smart contracts to assure the security of software engineering process update transactions. We are not at this time investigating how blockchains and smart contracts may be used as potential mechanisms that support the financial transactions or new business models for purchasing the services or products associated with a OA software system acquisition [Scacchi and Alspaugh 2016]. That is a topic for future research. Similarly, though blockchains and smart contracts are relatively new, they also entail their own set of vulnerabilities associated with their different technological implementations [Atzei, Bartoletti, Cimoli 2016] that must be addressed. Whether or how such vulnerabilities may manifest within acquisition processes is also a topic for future research.

Recommendations: Future extensions and new research elaborations

We have discussed the application of a blockchain system for coordinating and steering the evolution of an OA software system that is produced or integrated by a single party. But a blockchain system is by its nature a distributed system, and though its distributedness does not in itself give extra benefit in multi-producer, multi-integrator software ecosystems, clearly it is as effective in recording evolution and provenance in them, and is already adapted to the challenges of interactions with many parties.

Future research topic – cybersecurity threat meta-model formalization and codification

First, the social and technical threats and unintentional acts indicated earlier in Section 2.3 can form a basis for an *OA software cybersecurity threat meta-model*, based on the kinds of threats presented earlier in the examination of social, technical and unintentional threats to software supply chains. Here we summarize the outline of such a meta-model based on a comparative analysis of the three kinds of threats identified across more than kinds of identified software supply chain threats: threats of unauthorized access, threats of denial of authorized access, and threats of malicious software.

– Recognizing and Preventing Unauthorized Access Opportunities

These threats can enable the release, exposure, or exfiltration of data, user guides, software products, security access control mechanisms such as keys and licenses, and other secret or restricted information. The release may be intentional or unintentional, and may involve bribery, identity spoofing, hacks, man-in-the-middle attacks, updates, backups, and other technical or non-technical means.

OA software cybersecurity threat meta-model construction (unauthorized access):

disclosure/granted-access to unauthorized people of:

- information (secret; security weakness)
- user guides
- software products
    - permission to use; modify; redistribute
        - Note, these can result from IP/Security License evolution updates
- data storage (facility; repository; media)
- security access control mechanisms (keys; lists; containers; virtual machines; licenses)

intentionally or unintentionally, because of:

- non-technical reasons
    - delivery mistake or repudiation
    - bribery (coerced actors)
    - identity spoofing
- technical reasons (“evolution update transactions”)
    - hacking (unauthorized updates)
    - insertion of malicious code
    - man-in-the-middle delivery interception attack
    - denial of service attack
    - deletion/destruction
    - modification or destruction of security mechanisms or capabilities
        - data access control, backup, and transfer
        - software update access control and transfer

– Recognizing and Preventing Opportunities for Denial of Authorized User Access

These threats involve all the aspects of unauthorized access threats, but invert their perspectives by denying access to the same sorts of information and control to authorized people.

OA software cybersecurity threat meta-model construction (denied authorized access):

non-disclosure/denied-access to authorized people of:

- information (secret; security weakness)

- user guides
- software products
  - unable to use; modify; redistribute
    - Note, these can result from IP/Security License evolution updates
- data storage (facility; repository; media)
- security access control mechanisms (keys (exceed local legal limit); (lists; containers; virtual machines (modified/destroyed)); licenses)
- software product delivery/acceptance confirmation receipts

intentionally or unintentionally of because of:

- non-technical reasons
  - delivery mistake or repudiation
  - bribery (coerced actors)
  - identity spoofing
- technical reasons (“evolution update transactions”)
  - hacking (unauthorized updates)
  - insertion of malicious code
  - man-in-the-middle delivery interception-modification attack
  - denial of service attack
  - deletion/destruction
  - modification or destruction of security mechanisms or capabilities
    - data access control, backup, and transfer
    - software update access control and transfer

– Recognizing and Preventing Opportunities for Introduction of Malicious Software

The final category of threats involve malicious software elements including source code, externally sourced components, libraries and middleware, software connectors such as APIs and protocol handlers, build and packaging scripts, operating system protection mechanisms, storage devices and removable media, and corrupted or counterfeit data. They may be accomplished through the same technical and non-technical means as the first two categories of threats, and involve introduction, modification, or deletion of the elements in question.

OA software cybersecurity threat meta-model construction (introduction of malicious software):

disclosure/access to authorized people of

- malicious/corrupted software elements (counterfeit; unauthorized-modified (malicious code insertion)) via infection site:
    - application source code
    - outsourced or open source software components
        - standalone software systems (executable binaries; source code)
        - apps
        - widgets
        - build/packaging scripts
        - mashups
    - software libraries/middleware
    - software connectors
        - application program interfaces (APIs)
        - operating system/utility scripting
        - protocol handlers
        - database management systems
        - storage repositories
            - application (files; file systems)
            - software source/binary code (files; file systems (e.g., GitHub; SourceForge))
        - software buses
    - software containers
        - common software installation packages
            - Note, this also applies to compliance/validation testing software/data sets
        - packed (compressed), encrypted code for installation, unpacking and execution in computer memory
        - operating system protection mechanisms/capabilities (e.g., SELinux, SEAndroid-- security enhanced Linux, Android, from NSA)
        - virtual machines
        - storage devices
        - removable media

- corrupted/malicious licenses (IP/Security)
    - can also allow for corrupted/counterfeit data (facility; repository; media) elements

intentionally or unintentionally because of:

- technical reasons (“evolution update transactions”)
    - hacking (unauthorized updates)
    - insertion of malicious code
    - man-in-the-middle delivery interception or hijacking attack
    - denial of service attack
    - deletion/destruction
    - modification or destruction of security mechanisms or capabilities
        - data access control, backup, and transfer
        - software update access control and transfer

Future research topic – formalizing a domain-specific language and processing environment for specifying cybersecurity threat models and defensive security licenses as enactable smart contracts

In our prior research, we have called for a declarative domain-specific language (DSL) for specifying the obligations and rights incorporated into IP and security licenses for OA software [Alspaugh and Scacchi 2012, Scacchi and Alspaugh 2013a]. Now we see that such a DSL can be extended to incorporate software engineering process transactions using process modeling language like PML [Noll and Scacchi 2001, Scacchi 2001] or a similar notation, and that such extension is advantageous for managing OA software security system and engineering process challenges. The design and incorporation of these extensions into the DSL is thus a next step for us to research, develop and refine.

Next, we have also called for research and development of software obligations and rights management systems (SORMS) as a core capability for the DoD, government agencies, and other enterprises to help manage and improve their OA software system buying power [Scacchi and Alspaugh 2015, Scacchi and Alspaugh 2016]. We envision a SORMS that interprets and evaluates DSLs for software licensing as an essential tool for enterprises that manage OA software systems, such as found in most large organizations in industry, government, and Defense. Such DSL interpretation and execution will manipulate transactions to software bill of materials (BOM) technical data for the accumulating and evolving ISC, as well as for each PSC that is to be added/integrated via OA system evolution updates. Such transactions are intended to be subject to the rights, obligations, and countermeasures stipulated in OA software system licenses that conform to new cybersecurity meta-models outlined above.

As noted earlier, unauthorized updates to an ISC, whether the result of a social, technical or unintentional software supply chain threat, would be detected and defended against by the design of the SORMS that interprets system security licenses as smart contracts. These computational contracts would need to run continuously whenever a OA software system is being used in normal operations, as well as when the system is being intentionally updated. Such an approach is similar to the operation of any remote network/web server with a database management back-end server that normally operates continuously by design. The computational burden for such server operations is anticipated to be very modest, since the continuous computations are primarily checking ISC hash code values posted and maintained in the software BOM repository associated with the ISC. If/when intentional PSC updates are planned and executed, then remote validation of before and after ISC hash code functions reveal either accepted matches (thus valid update) or otherwise unaccepted or mismatch (indicating invalid update, thus the PSC is not installed, or the current ISC is marked as suspect, and identified as a candidate to be rolled back to a known valid ISC, perhaps indicating an unauthorized ISC update).

Thus, we call for effort to add capabilities that extend the SORMS to accommodate blockchain ledgers that manage and store software BOM repositories for both PSC to be installed and for accumulating and evolving ISCs, as decentralized or centralized databases, on which are enacted security licenses as smart contracts for automated handling software development and evolution process update transactions.

Conclusions

We sought to stimulate the development of innovative approaches to continuously assuring the cybersecurity of Open Architecture (OA) software system. We focused attention to exploring the potential for using blockchains and smart contract techniques, and how they can be applied to support acquisition efforts for software systems for OA command and control, or business enterprise (C2/B) systems. We further limited our focus to examining the routine software system updates to OA software configuration specifications that arise during the development and evolution processes arising during system acquisition. Our efforts described through our case study and related efforts thus denote a promising line of work in progress.

Much remains to be done, but the direction forward appears robust, productive, and likely to stimulate new innovations as a result of future research opportunities that we have recommended. We welcome questions and comments that identify possible oversights, as well as suggest complementary capabilities that enhance the potential of blockchain and smart contract tools, techniques, and technologies for continuously assuring the cybersecurity of software supply chains that support the development and evolution of modular, open architecture software systems as installed software configurations.

THIS PAGE INTENTIONALLY LEFT BLANK

References

Al Sabbagh, B. & Kowalski, S. (2015). A Socio-Technical Framework for Threat Modeling a Software Supply Chain, *IEEE Security and Privacy*, July-August, 30-39. Also see, X. Wang, B. Al Sabbagh, and S. Kowalski, “A Socio-Technical Framework for Threat Modeling a Software Supply Chain,” *Proc. Dewald Roode Workshop on Information Systems Security Research*, IFIP WG8.11/WG11.13, 2013, article 17.

Alberts C, Holler J, Wallen C. and Woody C. (2017). Assessing DoD System Acquisition Supply Risk, *CrossTalk: The Defense Software Engineering Journal*, 30(3), 4-9, May-June.

Alspaugh T.A, Asuncion H, and Scacchi W. (2013). The Challenge of Heterogeneously Licensed Systems in Open Architecture Software Ecosystems, in S. Jansen, S. Brinkkemper, and M. Cusumano (Eds.), *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry,* Edward Elgar Publishing, 103-120, Northampton, MA.

Alspaugh TA, Asuncion HU, and Scacchi W. (2009). Intellectual property rights requirements for heterogeneously-licensed systems. In *17th IEEE International Requirements Engineering Conference* (RE’09), pages 24–33, 2009.

Alspaugh TA, Asuncion HU, and Scacchi W. (2011) Presenting software license conflicts through argumentation. In *23rd International Conference on Software Engineering and Knowledge Engineering* (SEKE 2011), pages 509–514, 2011.

Alspaugh TA and Scacchi W. (2009). Heterogeneously-licensed system requirements, acquisition, and governance. In *Second International Workshop on Requirements Engineering and Law* (RELAW’09), pages 13–14, 2009.

Alspaugh TA and Scacchi W. (2012). Security Licensing, *Proc. Fifth Intern. Workshop on Requirements Engineering and Law*, 25-28, September 2012.

Alspaugh TA, Scacchi W, and Asuncion HA. (2010). Software Licenses in Context: The Challenge of Heterogeneously Licensed Systems, *J. Assoc. Info. Systems*, 11(11), 730-755.

Alspaugh TA, Scacchi W, and Kawai R. (2012). Software licenses, coverage, and subsumption. In *Fifth International Workshop on Requirements Engineering and Law (RELAW’12)*, pages 17–24, 25 Sep. 2012.

Atzei N, Bartoletti M, Cimoli T. (2016). *A Survey of Attacks on Ethereum Smart Contracts*, Cryptology ePrint Archive, Report 2016/1007, http://eprint.iacr.org/2016/1007

Berne (1979). *Berne Convention for the Protection of Literary and Artistic Works*, 1979.

Blockchain (2017). https://en.wikipedia.org/wiki/Blockchain, accessed 15 March 2017.

BOM (2017). *Software Bill of Materials*, https://en.wikipedia.org/wiki/Software_bill_of_materials accessed September 2017.

Brumaghin E, Gibb R, Mercer W, Molyett M, and Williams C (2017). CCleanup: A vast number of machines at risk. *Cisco TALOS Blog*, 18 September 2017.

Cherepanov A. (2017). Analysis of Telebots'Cunning Backdoor. *WeLiveSecurity.com*, https://www.welivesecurity.com/2017/07/04/analysis-of-telebots-cunning-backdoor/

Damianou N, Dulay N, Lupu E, and Sloman, M. (2001). The Ponder policy specification language. In *Int. Workshop on Policies for Distributed Systems and Networks*, pages 18–38, 2001.

(DISA) Defense Information Systems Agency. *Android 2.2 (Dell) Security Technical Implementation Guide* (STIG), 2011.

(DoD) Department of Defense, *Better Buying Power*, http://bbp.dau.mil/ accessed 25 May 2016.

(DoDGSA) Department of Defense and General Services Administration (2013). *Improving Cybersecurity and Resilience through Acquisition*, November 2013, accessed June 2015.

DuPont, Q. and Maurer, B. (2015). Ledgers and Law in the Blockchain, *King's Review*, 23 June 2015. http://kingsreview.co.uk/articles/ledgers-and-law-in-the-blockchain/

Ellison RJ, et al., (2010). *Evaluating and Mitigating Software Supply Chain Security Risks*, tech. report CMU/SEI-2010-TN-016, Software Eng. Inst., Carnegie Mellon Univ., 2010.

Endres-Niggemeyer B. (2013). The Mashup Ecosystem, in *Semantic Mashups: Intelligence Reuse of Web Resource*s, Springer, 1-50.

Ethereum (2017). https://en.wikipedia.org/wiki/Ethereum, accessed 15 March 2017.

Falliere N, Murchu LO, and Chien E. (2011). *W32.Stuxnet dossier*. Technical report, Symantec, 2011.

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

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

George A, Galdorisi G, Morris M, and O'Neil M. (2014). DoD Application Store: Enabling C2 Agility?, *Proc. 19th Intern. Command and Control Research and Technology Symposium*, Paper-104, Alexandria, VA, June 2014.

George A, Morris M, and O'Neil M. (2014). Pushing a Big Rock Up a Steep Hill: Lessons Learned from DoD Applications Storefront, *Proc. 11th Annual Acquisition Research Symposium*, Vol. 1, 306-317, Naval Postgraduate School, Monterey, CA.

Greenberg A. (2017). Software has a serious supply-chain security problem. *Wired*, Sept. 2017.

Guertin NH, Sweeney R, and Schmidt DC. (2015). How the Navy Can Use Open Systems Architecture to Revolutionize Capability Acquisition: The Naval OSA Strategy Can Yield Multiple Benefits. *Proc 12th Annual Acquisition Research Symposium,* Monterey, CA, NPS-AM-15-004, May 2015.

Guertin N and Womble B. (2012). Competition and the DoD Marketplace, *Proc. 9th Acquisition Research Symposium*. Vol. 1, 76-82, Naval Postgraduate School, Monterey, CA.

Henning, E (2011). Attack of the computer mouse. *The H Security*, 2011. http://h-online.com/-1270018 .

ISO/IEC (2005). International standard 27001.

Kendall F. (2015). *Implementation Directive for Better Buying Power 3.0,* 9 April 2015.

Kennedy M and Muttik I. (2011). *IEEE Taggant System*, https://standards.ieee.org/develop/indconn/icsg/taggant.pdf . Also see Software Taggant (2017). https://en.wikipedia.org/wiki/Software_taggant , accessed September 2017.

Loscocco PA, Smalley SD, Muckelbauer PA, et al. (1998) The inevitability of failure: The flawed assumption of security in modern computing environments. In *21st National Information Systems Security Conference* (NISSC'98), 1998.

Luo T, and Du W. (2011). Contego: Capability-based access control for web browsers. In *4th International Conference on Trust and Trustworthy Computing* (TRUST'11), 2011.

Mactal R & Spruill N. (2012). A Framework for Reuse in the DoN. *Proc. 9th Acquisition Research Symposium,* Vol.1, 149-164, Naval Postgraduate School, Monterey, CA.

Menn J (2017). Hackers compromised free CCleaner software, Avast's Piriform says. *Reuters*, 18 September 2017.

Noll J & Scacchi W. (2001). Specifying Process-Oriented Hypertext for Organizational Computing, *J. Network and Computer Applications,* 24(1):39-61.

Reed H, Benito P, Collens J, and Stein F. (2012). Supporting Agile C2 with an Agile and Adaptive IT Ecosystem, *17th Intern. Command and Control Research and Technology Symposium* (ICCRTS), Paper-044, Fairfax, VA, June 2012.

Reed H, Nankervis J, Cochran J, Parekh R, and Stein F. (2014). Agile, Adaptive IT Ecosystem: Results, Outlook, and Recommendations, *Proc. 19th Intern. Command and Control Research and Technology Symposium* (ICCRTS), Paper-011, Arlington, VA, June.

Riley M, Robertson J, and Sharpe A (2017). The Equifax Hack has the Hallmarks of State-Sponsored Pros, *Bloomberg Businessweek*, 29 September 2017, https://www.bloomberg.com/news/features/2017-09-29/the-equifax-hack-has-all-the-hallmarks-of-state-sponsored-pros

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

Sawers P. (2011). US Govt. plant USB sticks in security study, 60% of subjects take the bait. *The Next Web* (TNW), 2011.

Scacchi W. (2001). Redesigning Contracted Service Procurement for Internet-based Electronic Commerce: A Case Study, *J. Information Technology and Management*, 2(3), 313-334.

Scacchi W and Alspaugh TA. (2011). Advances in the Acquisition of Secure Systems Based on Open Architectures, *Proc. 8th Acquisition Research Symposium*, Vol. 1, Naval Postgraduate School, Monterey, CA.

Scacchi W and Alspaugh TA. (2012) Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems, *J. Systems and Software*, 85(7), 1479-1494, July.

Scacchi W and Alspaugh TA. (2013a) Advances in the Acquisition of Secure Systems Based on Open Architectures, in *Journal of Cybersecurity & Information Systems,* 1(2), 2-16, February 2013.

Scacchi W and Alspaugh TA. (2013b). Processes in Securing Open Architecture Software Systems, *Proc. 2013 Intern. Conf. Software and System Processes*, 126-135, San Francisco, CA, May 2013.

Scacchi W and Alspaugh TA. (2013c). Challenges in the Development and Evolution of Secure Open Architecture Command and Control Systems, *Proc. 18th*

Intern. Command and Control Research and Technology Symposium, Paper 098, Alexandria, VA, June 2013.

Scacchi W and Alspaugh TA. (2014). Achieving Better Buying Power through Cost-Sensitive Acquisition of Open Architecture Software Systems. *Proc 11th Annual Acquisition Research Symposium,* Monterey, CA, NPS-AM-14-C11P07R01-036, May 2014.

Scacchi W and Alspaugh TA. (2015). Achieving Better Buying Power through Acquisition of Open Architecture Software Systems for Web and Mobile Devices. *Proc 12th. Annual Acquisition Research Symposium,* Monterey, CA, NPS-AM-15-005, May 2015.

Scacchi W and Alspaugh TA. (2016). Achieving Better Buying Power for Mobile Open Architecture Software Systems Under Diverse Acquisition Scenarios. *Proc 13th Annual Acquisition Research Symposium,* Monterey, CA, SYM-AM-16-033, 163-183, May 2016.

Scacchi W and Alspaugh TA. (2017a). Issues in Development and Maintenance of Open Architecture Software Systems, *CrossTalk: The Defense Software Engineering Journal*, 30(3), 10-15, May-June.

Scacchi W and Alspaugh TA. (2017b). Cybersecure Modular Open Architecture Software Systems for Stimulating Innovation, *Proc. 14th Annual Acquisition Research Symposium*, SYM-AM-17-062, 316-334, Monterey, CA, April 2017.

Scacchi W and Alspaugh TA. (2017c). Identifying and Analyzing Cybersecurity Threats to Software Supply Chains. Technical Report, Institute for Software Research, UC Irvine, (in progress).

Scacchi W and Alspaugh TA.(2017d). Security Licenses as Smart Contracts for Securing the Software Supply Chains. Technical Report, Institute for Software Research, UC Irvine, (in progress).

Seacord RC (2008). *CERT C Secure Coding Standard*. Addison-Wesley, 2008.

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

Smart Contracts (2017). https://en.wikipedia.org/wiki/Smart_contract, accessed 25 May 2017.

Spencer R, Smalley S, Loscocco P, Hibler M, Andersen D, and Lepreau J (1999). The Flask security architecture: System support for diverse security policies. In *8th. USENIX Security Symposium* (SSYM'99), pages 11–11.

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

(UKCSA) UK Government Chief Scientific Adviser (2016). *Distributed Ledger Technology: Beyond Block Chain*. Government Office for Science, London. https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/492972/gs-16-1-distributed-ledger-technology.pdf

US-CERT (2017). Petya ransomware. Alert TA17-181A, 28 July 2017, https://www.us-cert.gov/ncas/alerts/TA17-181A

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

Wang X, Al Sabbagh B, and Kowalski S (2013). A Socio-Technical Framework for Threat Modeling a Software Supply Chain, *Proc. Dewald Roode Workshop on Information Security Research, IFIP WG8.11/WG11.13,* Paper 17*.*

Womble B, Schmidt W, Arendt M, and Fain T. (2011). Delivering Savings with Open Architecture and Product Lines, *Proc. 8th. Acquisition Research Symposium*, Vol. 1, 8-13, Naval Postgraduate School, Monterey, CA.

Xen (2017). *Xen Hypervisor*. http://xen.org/products/xenhyp.html .

Zheng D and Carter W. (2015). *Leveraging the Internet of Things for a More Efficient and Effective Military*, Center for Strategic & International Studies. Washington, DC, September 2015.

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC
POLICY
NAVAL POSTGRADUATE SCHOOL
555 DYER ROAD, INGERSOLL HALL
MONTEREY, CA 93943

www.acquisitionresearch.net