Cybersecure Modular Open Architecture Software Systems for Stimulating Innovation

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 to 1998, he was on the faculty at the University of Southern California. In 1999, he joined the Institute for Software Research at UC Irvine. He has published more than 200 research papers and has directed more than 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 for the Eighth IFIP International Conference on Open Source Systems. [wscacchi@ics.uci.edu]

Thomas A. 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. He also worked as a computer scientist at the Naval Research Laboratory on the Software Cost Reduction, or A-7, project. [thomas.alspaugh@acm.org]

Abstract

Our interest is to stimulate the development of innovative approaches to continuously assuring the cybersecurity of open architecture (OA) software systems. We focus on exploring the potential for using blockchains and smart contract techniques and how these techniques 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 discuss 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 the 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 the expanded scope of cybersecurity assurance of not just the delivered OA C2/B software systems, but also in the engineering processes which create, transform, or otherwise update technical data that is central to the acquisition of OA software systems.

Overview

How might we stimulate the development of innovative approaches to continuously assuring the cybersecurity of open architecture (OA) software systems? This is the acquisition research challenge we are addressing. In particular, we are interested in investigating innovations that represent either incremental improvements or substantial departures from our current acquisition practices for 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. We seek to stimulate significant innovations that employ emerging concepts and technologies to problems observable in the acquisition, development, and evolution of modern C2/B systems.

Problem

The particular problem we investigate here is how best to develop and demonstrate a new conceptual approach to providing continuous cybersecurity assurance (DoD & GSA, 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."

Solution

The innovations 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.

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?

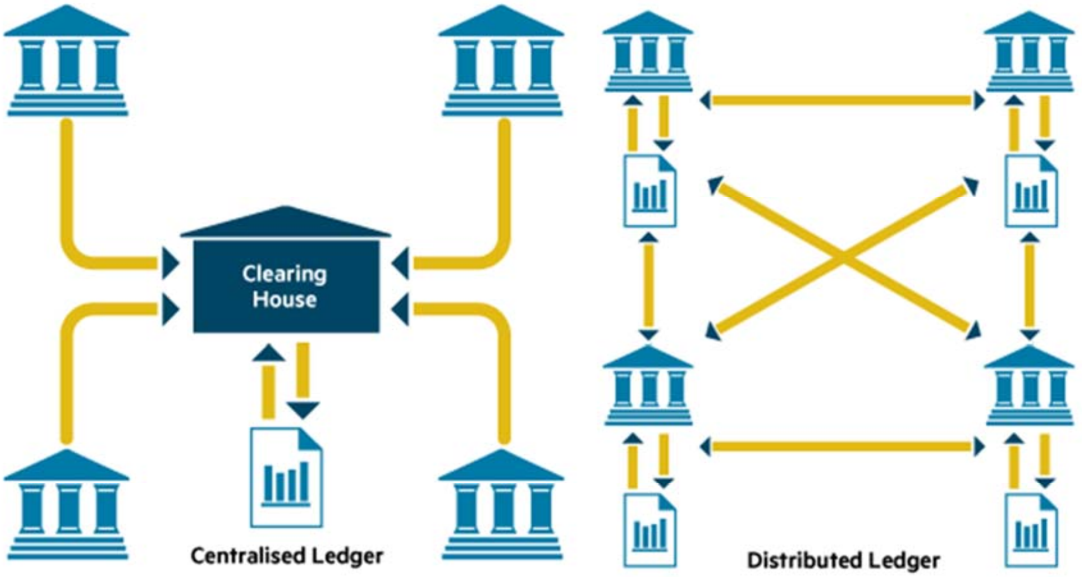
Based on prior studies of issues and challenges arising in the development and evolution of OA software systems for C2/B system applications (Guertin, Sweeney, & Schmidt, 2015; Scacchi & Alspaugh, 2012–2017; Womble et al., 2011), we have already drawn attention to technical problems that arise in the 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.

Background: Blockchains and Smart Contracts as Ledgers and Contractual Agreements for Tracking and Managing Transactions

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 & 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. 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," n.d.). 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,” n.d.). Figure 1 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 1. Traditional Ledger Network (Left) and Decentralized Blockchain Ledger Network (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,” n.d.). 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 are similarly 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. 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 denote which transactions to process (where, when, how, and for what parties) in what order ("Smart Contracts," n.d.). 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 have traditionally 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 only 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 the software development and evolution processes of large, long-term software acquisition efforts.

Blockchains are being extended to accommodate smart contracts that allow for the formation of virtual, decentralized autonomous organizations that act to govern, enforce, and assure the integrity and validity of complex or idiosyncratic blockchain update transactions ("Smart Contracts," n.d.). For example, multi-party agreements whereby two or more program offices or other enterprises can act to share the procurement costs of a new C2/B system application or component of mutual interest to the participating parties (Reed et al., 2012; Reed et al., 2014; Scacchi & Alspaugh, 2015). 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?

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? 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 those 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. Such a specification is a kind of technical data to be managed with an acquisition effort.

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 using 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 and whether these are apps or updates that were innocently installed but not recorded in the ledger or attacks—maliciously injected software for some nefarious purpose, which would not be 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 previously 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 in the hashcode value associated with the current system when last updated and verified by miners, and this hashcode may reveal whether the ISC specification copy's hashcode matches the one checked during audit or subsequent miner verification activities. If the hashcode 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 later, 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 hashcode value (a long, non-guessable string of characters that is generated within the blockchain system) 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 mismatch, then a rollback may be triggered that reverts the ISC on the machine to a previously trusted ISC, and then removes, deprecates, or flags the unverified ISC as suspect, along with distributing a 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 ensuring access and update rights encoded in a software system's security license (Alspaugh & Scacchi, 2012).

An Example Ledger, Transaction, Smart Contract Implementation System

Ethereum is being used to implement smart contracts, transactions, and a blockchain ledger ("Ethereum," n.d.). 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 U.S. 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 who 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.

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 2. 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 be deployed only by request or authorization when functionally similar system version updates are made available (Scacchi & Alspaugh, 2013a, 2015, 2016, 2017).

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 the following section.

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 & 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.

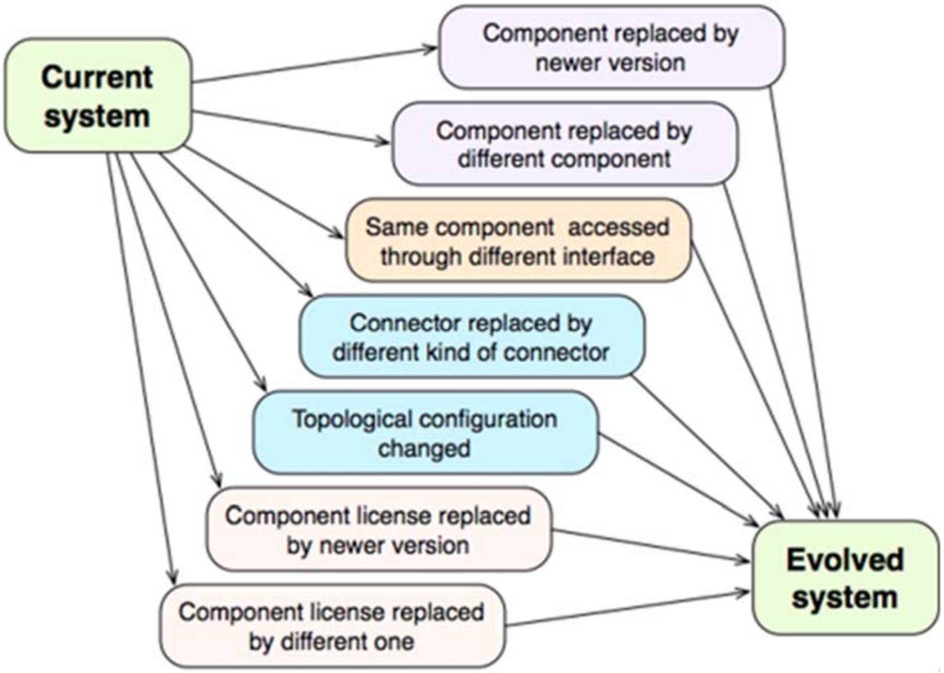

Current system
Component replaced by newer version
Component replaced by different component
Same component accessed through different interface
Connector replaced by different kind of connector
Topological configuration changed
Component license replaced by newer version
Component license replaced by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

Figure 2. Seven Types of Software Evolution Update Transactions
(Scacchi & Alspaugh, 2013)

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 may 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 to 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 & 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 the software evolution processes. As described previously, 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 & Alspaugh, 2013b, 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 & Carter, 2015), as we have addressed before (Scacchi & Alspaugh, 2015, 2016).

Blockchain ledgers serve to verify in a decentralized manner the proper sequencing of valid transactions for a user/device account. Such an account operates like a personal bank account that can be used to deposit and withdraw funds (e.g., through account transactions associated with a debit/credit card that is 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., the party who receives 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 the corresponding virtual private network's blockchain ledger in ways that maintain device/user autonomy, but reveal 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 battleship or a multi-ship flotilla (or, alternatively, an aircraft flight wing, 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 a 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.

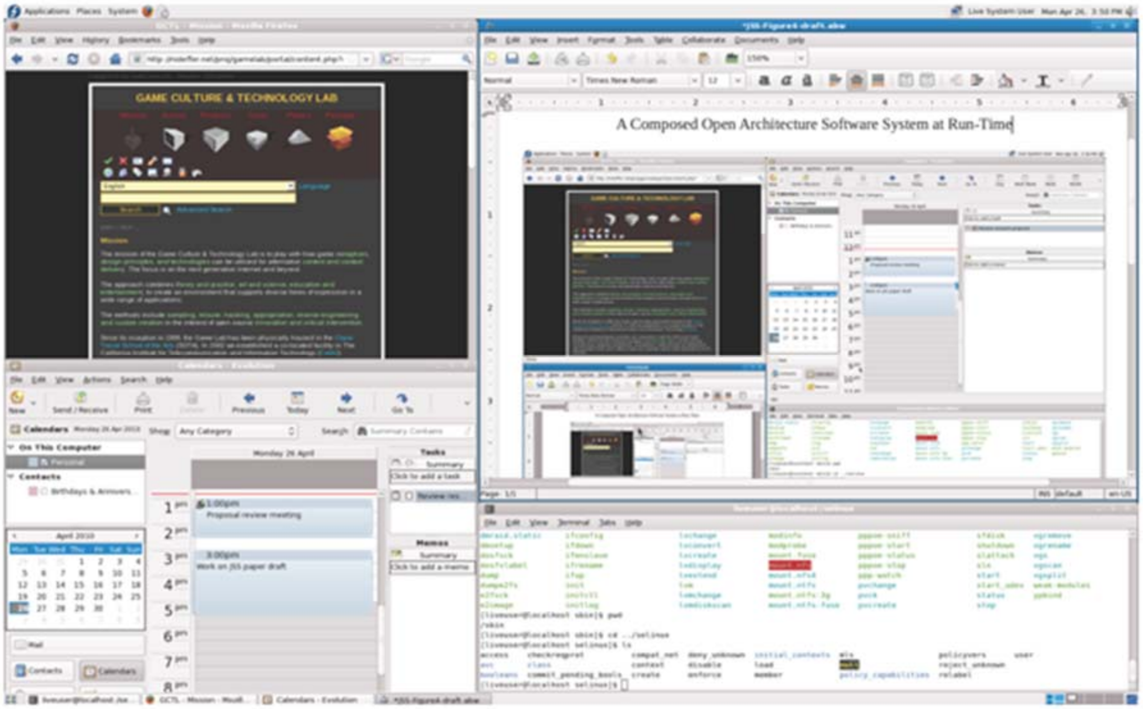
Consider a desktop PC with apps/widgets acquired from either a restricted-access enterprise-specific app store, a Defense app store (George et al., 2014; George, Morris, & O'Neil, 2014), or else from a public-access app store or OSS component repository. Web browser-based apps like cloud-based word processors, 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 & 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 & Alspaugh, 2015, 2016).

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 a PC operating system 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 3. 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 similar 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 6.

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 3 to an evolved ISC seen in Figure 6. Figure 3 corresponds to its ISC model visualized in Figure 4, 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 hashcode 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.

For example, we may elect to use a predefined 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 that the ISC blockchain hashcode 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); (4) replace ISC interconnection topology with the evolved ISC; (5) send request to miners to independently compute and verify the evolved ISC specification hashcode value on the target machine/platform denotes the ISC and associated meta-data they independently build to compute the evolved ISC hashcode; (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; and (e) perform 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 5, the configuration model of the evolved OA system still incorporates the same kind of components as the current system model (shown in Figure 4), 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).


Applications Places System
GAME CULTURE & TECHNOLOGY LAB
A Composed Open Architecture Software System at Run-Time
File Edit View Terminal Tabs Help

```
[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 3. Current Deployed OA ISC Corresponding to Figure 4, Utilized by End-Users

Note. Firefox Web Browser (Upper Left), Evolution Calendar (Lower Left), AbiWord Word Processor (Upper Right), and Fedora/Linux Desktop Operating System Platform (Lower Right)

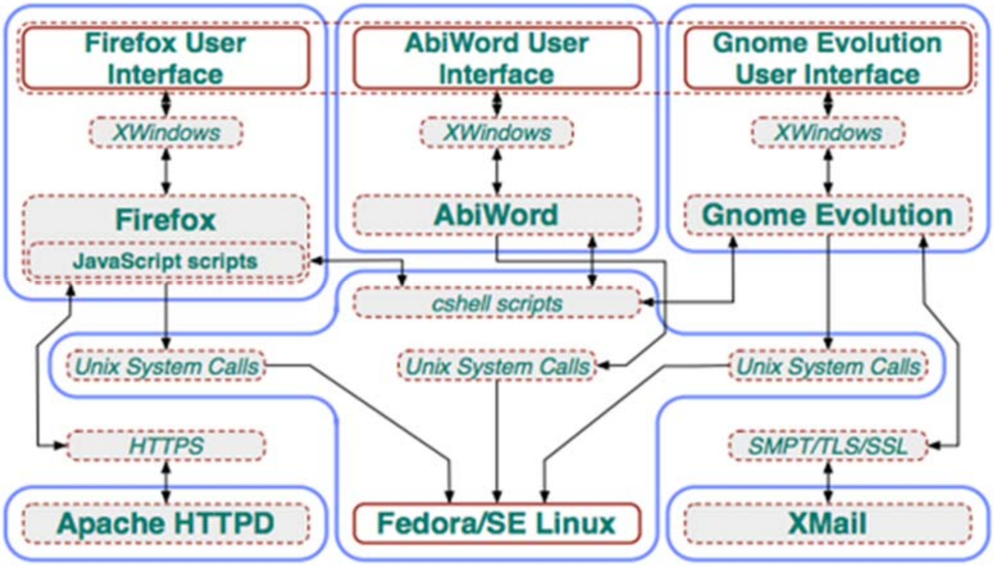


Figure 4. Current ISC Specification for OA C2/B System
(Scacchi & Alspaugh, 2013, 2017)
Note. This is the current ISC specification for an OA C2/B system within security containers at build-time, intended to denote a record on the blockchain for which components need to be included during integration (and testing) of the software components and code APIs within the released and deployed ISC.

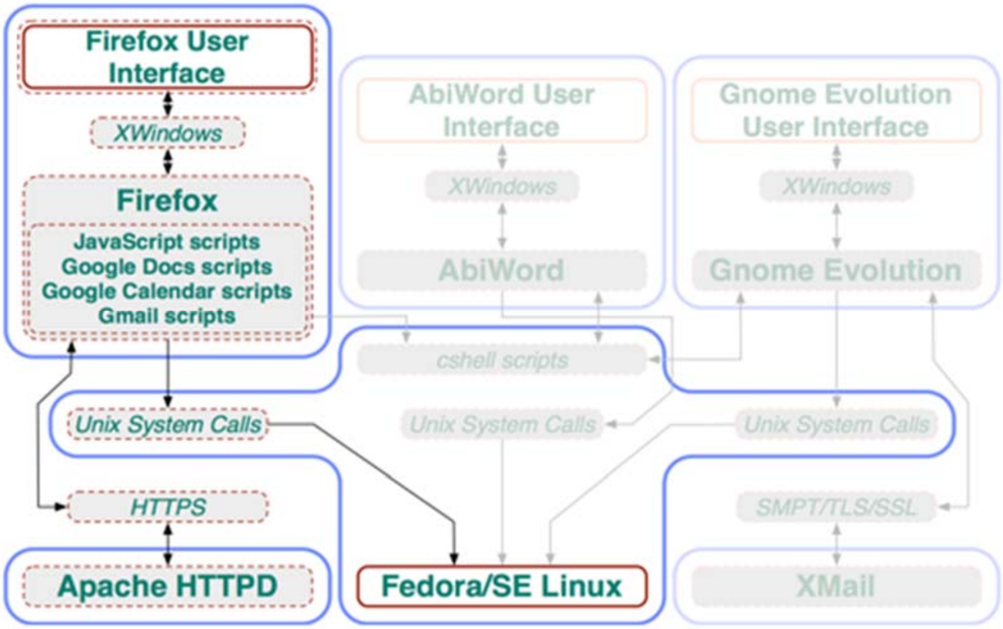


Figure 5. The Evolved OA ISC Specification at Build-Time
Note. The topology of the ISC has evolved, including where now legacy components have been deprecated and likely marked for eventual removal, so as to eliminate any residual vulnerability pathway still present.

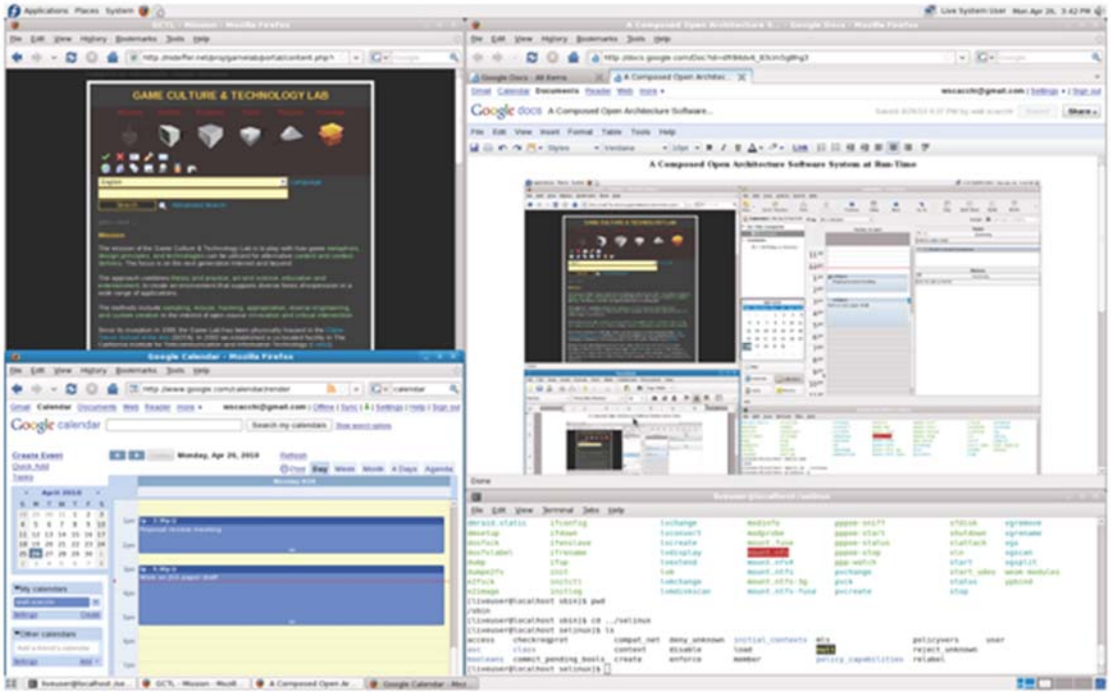


Figure 6. Evolved OA ISC Corresponding to Figure 5, Installed for Utilization by End-Users

Note. Firefox Web Browser as Before (Upper Left), Google Calendar (Lower Left), Google Docs (Upper Right), and Fedora/Linux Operating System Platform as Before (Lower Right)

The transformation of the current system in Figure 3 and Figure 4 to the evolved system in Figure 5 and Figure 6 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.

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.

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 2 also classify comparable types of attacks on OA systems during their development, build, deployment, and run time processes. 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 are clearly possible [Scacchi & Alspaugh, 2012, 2015, 2016].) 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 hashcode value of the compromised system, compared to the current system's provenance established 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 attacks during development, build, and deployment, because the context is more complex. Here 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, transactions, and smart contracts 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 & 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.

Future Extensions and Elaborations of This Approach

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 it is already adapted to the challenges of interactions with many parties.

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 & Scacchi, 2012; Scacchi & Alspaugh, 2013a). Now we see that such a DSL can be extended to incorporate software engineering process transactions using a process modeling language like PML (Noll & 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.

Last, 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 & Alspaugh, 2015, 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 those found in most large organizations in industry, government, and defense. Thus, we call for effort to add capabilities that extend the SORMS to accommodate blockchain ledger repositories, as decentralized or centralized databases, on which are enacted smart contracts for handling software development and evolution process update transactions.

Conclusions

In this paper, we sought to stimulate the development of innovative approaches to continuously assuring the cybersecurity of open architecture (OA) software systems. We focused on 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 and productive. We welcome questions and comments that identify possible oversights, and we suggest complementary capabilities that enhance the potential of blockchain and smart contract tools, techniques, and technologies for continuously assuring the cybersecurity of modular open architecture software systems.

References

Alspaugh, T. A., Asuncion, H., & 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* (pp. 103–120). Northampton, MA: Edward Elgar.

Alspaugh, T. A., & Scacchi, W. (2012, September). Licensing security. In *Proceedings of the Fifth International Workshop on Requirements Engineering and Law* (pp. 25–28).

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

Atzei, N., Bartoletti, M., & Cimoli, T. (2016). *A survey of attacks on Ethereum smart contracts*. Retrieved from http://eprint.iacr.org/2016/1007.pdf

Blockchain. (n.d.). In *Wikipedia*. Retrieved March 15, 2017, from https://en.wikipedia.org/wiki/Blockchain

DoD & GSA. (2013, November). *Improving cybersecurity and resilience through acquisition*. Retrieved from https://www.gsa.gov/portal/getMediaData?mediaId=185367

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

Endres-Niggemeyer, B. (Ed.). (2013). The mashup ecosystem. In *Semantic mashups: Intelligence reuse of web resources* (pp. 1–50). Springer.

Ethereum. (n.d.). In *Wikipedia*. Retrieved March 15, 2017, from https://en.wikipedia.org/wiki/Ethereum

George, A., Galdorisi, G., Morris, M., & O'Neil, M. (2014, June). DoD application store: Enabling C2 agility? In *Proceedings of the 19th International Command and Control Research and Technology Symposium* (Paper 104). Alexandria, VA.

George, A., Morris, M., & O'Neil, M. (2014). Pushing a big rock up a steep hill: Lessons learned from DoD applications storefront. In *Proceedings of the 11th Annual Acquisition Research Symposium* (Vol. 1, pp. 306–317). Monterey, CA: Naval Postgraduate School.

Guertin, N. H., Sweeney, R., & Schmidt, D. C. (2015, April). How the Navy can use open systems architecture to revolutionize capability acquisition: The Naval OSA strategy can yield multiple benefits. In *Proceedings of the 12th Annual Acquisition Research Symposium* (Vol. 1, pp. 107–116). Monterey, CA: Naval Postgraduate School.

Noll, J., & Scacchi, W. (2001). Specifying process-oriented hypertext for organizational computing. *Journal of Network and Computer Applications, 24*(1), 39–61.

Reed, H., Benito, P., Collens, J., & Stein, F. (2012, June). Supporting agile C2 with an agile and adaptive IT ecosystem. In *Proceedings of the 17th International Command and Control Research and Technology Symposium* (Paper 44). Fairfax, VA.

Reed, H., Nankervis, J., Cochran, J., Parekh, R., & Stein, F. (2014, June). Agile, adaptive IT ecosystem: Results, outlook, and recommendations. In *Proceedings of the 19th International Command and Control Research and Technology Symposium* (Paper 11). Arlington, VA.

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

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

Scacchi, W., & Alspaugh, T. (2013a, February). Advances in the acquisition of secure systems based on open architectures. *Journal of Cybersecurity & Information Systems, 1*(2), 2–16.

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

Scacchi, W., & Alspaugh, T. (2014). Achieving Better Buying Power through cost-sensitive acquisition of open architecture software systems (NPS-AM-14-C11P07R01-036). In *Proceedings of the 11th Annual Acquisition Research Symposium*. Monterey, CA: Naval Postgraduate School.

Scacchi, W., & Alspaugh, T. (2015). Achieving Better Buying Power through acquisition of open architecture software systems for web and mobile devices (NPS-AM-15-005). In *Proceedings of the 12th Annual Acquisition Research Symposium*. Monterey, CA: Naval Postgraduate School.

Scacchi, W., & Alspaugh, T. (2016). Achieving Better Buying Power for mobile open architecture software systems under diverse acquisition scenarios (SYM-AM-16-033). In *Proceedings of the 13th Annual Acquisition Research Symposium* (pp. 163–183). Monterey, CA: Naval Postgraduate School.

Scacchi, W., & Alspaugh, T. A. (2017). Issues and challenges in the operations and maintenance of open architecture software systems. *CrossTalk: The Defense Software Engineering Journal,* 10–15.

Smart Contracts. (n.d.). In *Wikipedia*. Retrieved March 25, 2017, from https://en.wikipedia.org/wiki/Smart_contract

Womble, B., Schmidt, W., Arendt, M., & Fain, T. (2011). Delivering savings with open architecture and product lines. In *Proceedings of the Eighth Annual Acquisition Research Symposium* (Vol. 1, pp. 8–13). Monterey, CA: Naval Postgraduate School.

Zheng, D., & Carter, W. (2015). *Leveraging the internet of things for a more efficient and effective military.* Center for Strategic & International Studies. Retrieved from https://www.csis.org/analysis/leveraging-internet-things-more-efficient-and-effective-military

Acknowledgments

The research described in this report was supported by grant #N00244-16-1-0053 from the Acquisition Research Program at the Naval Postgraduate School, Monterey, CA. No endorsement, review, or approval implied. The concepts, topics, and materials presented are the sole responsibility of the authors.

Acquisition Research Program
Graduate School of Business & Public Policy
Naval Postgraduate School
555 Dyer Road, Ingersoll Hall
Monterey, CA 93943

www.acquisitionresearch.net