Secure Software Development – Models, Tools, Architectures and Algorithms

Aspen Olmsted
College of Charleston
Department of Computer Science, Charleston, SC 29401
e-mail: olmsteda@cofc.edu

***Abstract*— Secure software development is a process which integrates people and practices to ensure application Confidentiality, Integrity, Availability, Non-Repudiation, and Authentication (CIANA). Secure software is the result of a security- aware software development process in which CIANA is established when an application is first developed. Current secure software development lifecycles are simply old software development lifecycles with security training prepended to the traditional development steps and an incident response process appended to the lifecycle. To solve our application cyber-security issues, we need to develop the models, tools, architectures, and algorithms that support CIANA on the first day of a development project.**

Keywords-Cyber-security; Software Engineering; CRM

I. INTRODUCTION

In this work, we investigate the problem of developing software that is built to provide the security required in our modern, connected world. Secure software development is the process involving people and practices that ensure application Confidentiality, Integrity, Availability, Non-Repudiation, and Authentication (CIANA). Secure software is the result of a security aware software development processes where CIANA is established when an application is first developed.

Current secure software development lifecycles (SSDLC) are just old software development lifecycles (SDLC) with a security training prepended before the traditional development steps and an incident response process append to the end of the lifecycle. To solve our application cyber-security issues, we need to develop the models, tools, architectures, and algorithms that support CIANA on the first day of a development project.

The organization of the paper is as follows. Section II describes the related work and the limitations of current methods. In Section III, we document student work from our lab in the creation of algorithms and architectures that provide consistency, availability, and partition tolerance for distributed systems. Section IV looks at algorithms the lab has developed to provide correctness guarantees for the integration of heterogeneous systems. Section V explores our solutions for authenticating autonomous processes and securing the communication between them. Section VI analyzes the lab's solutions for securing code and data in an operating system. In Section VII, we share our additions to UML modeling to move the awareness of potential system vulnerabilities to an earlier point in the software development life-cycle. Finally, in Section VIII, we look at ways to reduce the software development cost through the use of cloud architectures. We conclude and discuss future work that needs to be done to advance our algorithms, architectures, tools, and modeling in Section IX.

II. RELATED WORK

For many years, software engineering firms followed an SDLC that consisted of five steps: requirements gathering, solution design, implementation, testing, and maintenance. Many new SSDLCs have evolved with the goal of helping software developers write software with fewer vulnerabilities. Microsoft created the Security Development Lifecycle [1] as a recommended solution to a more SDLC. In the Microsoft recommendations, a preliminary training phase is introduced to teach users to not only distrust data from external sources but also to understand typical vulnerabilities found in software applications. In the testing phase, they recommend using penetration testing software to ensure typical secure programming mistakes are caught. At the end of the development lifecycle, they recommend implementing a response system to address the software once a vulnerability has been found. Our work attempts to be more proactive in developing the models, tools architectures, and algorithms the developers need to guarantee vulnerabilities are discovered and addressed earlier in the development lifecycle.

Over the last decade, many books have been written to help developers understand the technical programming solutions to two standard problems:

1. SQL Injection – A vulnerability in an application through which a malicious user can execute malicious SQL statements against the application's back end data store.
2. Cross Site Scripting – A vulnerability in an application through which a malicious user can execute client side JavaScript inside a page of the application.

One such book is Edmund's recent book *Securing PHP Applications* [2]. In each of these books, algorithms which sanitize user input that may be coming from user forms, cookies, or even the back-end database are explained in detail. The basic premise these books espouse is trust no-one. We attempt to give the developer some trust in addition to their programming repertoire already consisting of the models, tools, architectures, and algorithms to guarantee security in the development process.

Walden, Doyle, Lenhof and Murray [3] studied whether the variation in vulnerability density is greater between languages or between different applications written in a single language by comparing eleven open source web applications written in Java with fourteen such applications written in PHP. To compare the languages, they created a Common Vulnerability Metric (CVM) which represents the count of four vulnerability categories common to both languages. Our work

 ISBN: 978-1-61208-582-1

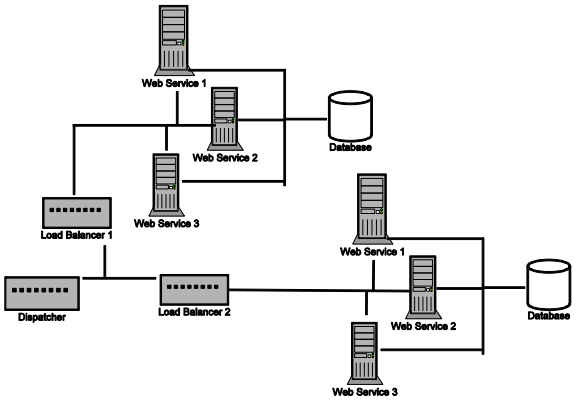

Figure 1. WS Farm with Buddy System.

here looks to find common vulnerabilities in enterprise applications and provide solutions to those vulnerabilities.

III. DISTRIBUTED CONSISTENCY, AVAILABILITY AND PARTITION TOLERANCE

Modern web-based transaction systems need to support many concurrent clients consuming a limited quantity of resources. These applications are often developed using a Service Oriented Architecture (SOA). SOA supports the composition of multiple web services (WSs) to perform complex business processes. SOA applications provide a high-level of concurrency; we can think of the measure of concurrency as the availability of the service to all clients requesting services. Replication of these services and their corresponding resources increases availability. Unfortunately, designers sacrifice consistency and durability to achieve this availability. The CAP theory [4] [5] states that distributed database designers can achieve at most two of the following properties: consistency (C), availability (A), and partition tolerance (P). Distributed database designers often relax the consistency requirements under its influence.

Our proposed system [6] has three benefits: it decreases the risk of losing committed transactional data in the event of a site failure, increases consistency of transactions, and increases the availability of "read" requests. The three main components of our proposed system are 1) Synchronous Transactional Buddy System, 2) Version Master-Slave Lazy Replication, and 3) Serializable Snapshot Isolation Schedule.

Our solution [6] adopts the WS-Farm (WSF) architecture (Figure 1) to allow the system to provide the features iterated above. Transactions arrive at the dispatcher at the TCP/IP level 7 allowing the dispatcher to use application specific data for transaction distribution and buddy selection. The dispatcher also receives the requests from clients and distributes them to the WS clusters which each contain a load balancer, a single database, and replicated services. The load balancer receives the service requests from the dispatcher and distributes them among the service replicas. Within a WS cluster, each service shares the same database, and database updates among the clusters are propagated using lazy replication propagation [6].

This method of propagation is vulnerable to a loss of updates in the event of a database server failure, though [6]. If a server failure occurs after the transaction has committed, but before the replica updates are initiated, the updates are lost. To guarantee data persistence even in the presence of hardware failures, we propose to form strict replication between pairs of replica clusters "buddies." In this method of replication, at least one replica in addition to the primary replica is updated and, therefore, preserves the updates.

After receiving a transaction, the dispatcher picks the two clusters, chosen by versioning history, to form the buddy-system. The primary buddy (b1) receives the transaction along with its buddy's (b2) IP address. The primary buddy (b1) becomes the coordinator in a simplified commit protocol between the two buddies. Both then buddies perform the transaction and commit or abort together. The dispatcher maintains metadata about the freshness of data items in the different clusters in addition to incrementing the version number for each data item after it has been modified. Any two service providers in two different clusters with the latest version of the requested data items can be selected as a buddy. Note, that the databases (DBR) maintained by the two clusters must have the same version of the requested data items but may not for the other data items.

IV. HETEROGENEOUS SYSTEM INTEGRATION

Enterprise transaction processing systems support several different use cases to fulfill the entire set of requirements of an organization. An organization will partition an enterprise system at the department level for several different reasons. Two of these reasons are to simplify the functional model and to enable geographic proximity to the users entering the transactional data.

The result of the departmental partitioning is a duplication of data across departmental systems, and the management of this duplication is a difficult problem. Often, an organization will enter this data manually in each local system. The organization is then forced to tolerate the data inconsistencies that come from the difference not only in human interpretation of the source data but also transcription differences.

In our previous work [7], we investigated the problem of providing guarantees for heterogeneous system integration. We proposed a set of strong properties: Fresh, Atomic, Consistent and Durable (FACD), which will deliver correct results when held in the integration transaction. The strong properties support an integration technique called Continuous, Consistent, Extract, Translate, and Load (CCETL). CCETL consumes UML class diagrams to identify transactional membership of the data elements that make up the integration. CCETL transforms the hierarchical relationships using a version of the topological sort that maintains a navigation path from the original UML classes.

 ISBN: 978-1-61208-582-1

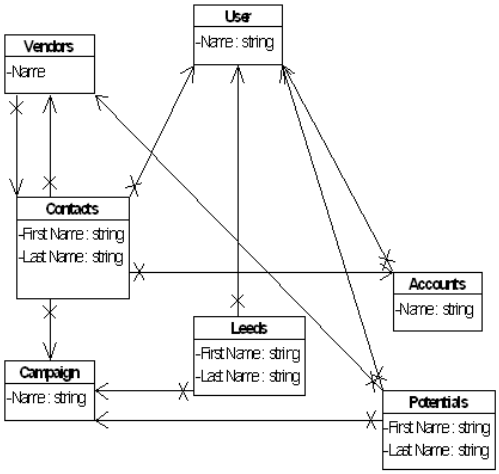

Figure 2. Cyclical UML Class Diagram.

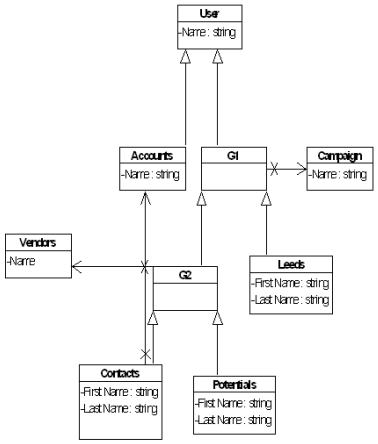

Figure 3. Acyclic Sorted UML Class Diagram.

The CCETL approach guarantees ACID properties up to the level of snapshot isolation between systems supporting a continuous integration.

The example application for CCETL used a collection of Zoho web-service and a back-office ERP solution for Cultural Arts Organizations named Tessitura [8]. Tessitura transactions include patron donations and ticket purchases. The Zoho web services [9] provide a timestamp on every entity record modification. This timestamp is used to identify all records changed since the last execution of the integration.

For this project, we choose to use a sub-model of the Zoho CRM service. Zoho CRM is a software-as-a-service product for managing customer data such as biographical data, emails, phone calls, etc. We choose Zoho for the project because the Zoho CRM product provides web services that allow user-defined data queries against all the available entity objects. Figure 2 shows a UML diagram of a subset of the web-services provided by Zoho. Each web service represents a coarse-grained entity object. The diagram shows the navigation knowledge of each web-service with respect to other web services. The associations form a directed cyclical-graph.

There are two ways to identify transactional data: intercept original transactions synchronously or reform transactions asynchronously from the original transaction. To intercept the original transaction synchronously, we need an application hook to inform the integration when a transaction is taking place. An example of this application hook is available in Oracle Forms [10]. Synchronous integration increases the latency of the original transaction.
To reform a transaction asynchronously from the original transaction, we need to identify what data changed in the original transaction. To identify which records make-up a transaction, CCETL includes all associated records modified along with the parent record. This identification requires an ordering of the original UML diagram Figure 2. In our previous work [7] we provide an algorithm that de-cycles and sorts the original graph.

Figure 3 shows a version of the original UML diagram from Figure 2 with cycles removed and sorted. The algorithm inserts mock objects when there are identical inbound edges into a node. The addition of the mock objects reduces the branches in the path of the UML graph.

We ran the integration two ways: integrate on a record by record change (Record Integration) and use CCETL. (Snapshot Integration). We ran the two integration techniques with transaction sizes in blocks of 100 up to 1000. The snapshot isolation method provided much higher throughput and provided isolation guarantees at snapshot isolation. The record integration method was slower and only provided isolation at the read committed isolation level. The higher latency and lower consistency stems from dealing with a single record at a time.

V. PROCESS AUTHENTICATION AND COMMUNICATION

Authentication is used to verify that a specific user or process is who they say they are and is one of the major domains in cyber security research. Unfortunately, autonomous process authentication is a neglected segment of this domain. The autonomous processes are often native operating system services, but sometimes the autonomous process is a part of a larger enterprise application where the process needs access to different resources unavailable to the user who is operating the application. In this case, the process needs different credentials. The resources protected fall into three categories: operating system files, data and process execution. Operating system files are traditionally secured based on the user logged into the operating system. Permissions can be discreetly assigned to the user or inherited from a user's group membership. Data permissions are normally managed by a relational database system. In the database system, access is granted to tables, columns and tuples in the database based on the user's credentials or the user's group membership. The permission to launch

 ISBN: 978-1-61208-582-1

processes is often guarded by the operating system based on the logged in user and the user's group membership.

There are four standard ways we authenticate users:

- Something you know – In this form of authentication, a user or process must know a secret. The typical secret used in authentication is a combination of a user and a password.
- Something you have – In this form of authentication, a user or process must have access to a physical entity. The typical example is a token that is sent to an SMS number. If the user has their registered phone and can receive SMS messages then only they can enter back the one time generated token. This form of authentication is not typically used with autonomous processes on servers because an operator with a mobile device is not typically on the server's console. Autonomous processes on mobile devices with SMS service can use this technique to validate that a user has the phone, but a server process does not typically have SMS support. If they do have SMS support, then the process is typically using a virtual SMS service which would no longer be something to which the process has access.
- Something about you - In this form of authentication, a unique characteristic is used to validate access. Examples include retina scans, fingerprint readers, and facial recognition. This form of authentication when dealing with a human operator tends to be the strongest form of authentication, but it is not used with process authentication as processes do not have these characteristics.
- Someplace you are – In this form of authentication, the address where the user or process is located controls access to the resources. Examples in this category include a range of IP addresses or the geographic longitude and latitude points where a machine may be operating.

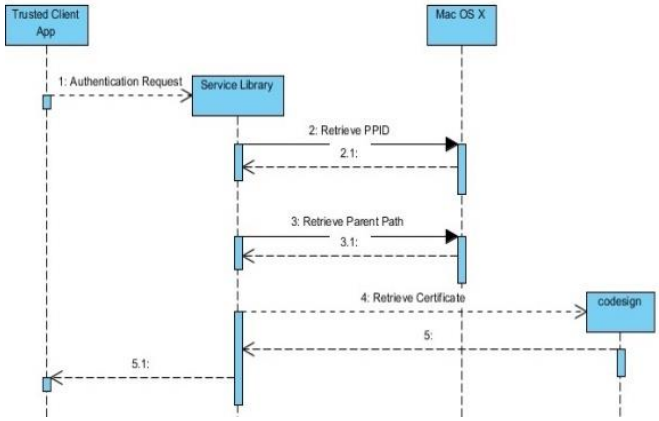

Figure 4. Sequence Diagram of Application Authentication Process.

In our previous work [11], we add autonomous process authentication in a limited environment. To add "something about you" security for an autonomous process, we investigated verifiable properties of an application. These properties need to validate the process is not a malicious user or a different process posing as the valid process. In this work, our solution uses the security certificate used to code-sign an application that is listed in the Mac Apple Store [12]. This certificate is not applied to the application for the validation purpose we propose, but it works quite nicely. The certificate is signed by Apple to ensure that no malicious user has changed the application code. Unlike in PKI, where a certificate can be signed by many different trusted third parties, the Mac Apple Store certificates are only signed by Apple, Inc. Algorithm 1 shows the algorithm we use to extract the certificate and validate the application. The current application requires a native Mac OS X application signed by the Apple Mac Store. Our service is a database service written as a native Mac OS X application. When the 3rd party application forks to our service app, we can retrieve the process id of the external application. With the process id, we can determine the operating system path of the application that is calling our service. From this path, we can validate the certificate. The Mac OS X operating system includes a utility called code-sign [13] that allows you to retrieve and verify the signature on an application. Our algorithm uses this utility in the final step to verify the process is the process it says it is. Figure 4 shows the flow of the algorithm in a UML sequence diagram. Our current work looks to leverage this work for inter-process communication.

VI. SECURE DATASTORE AND CODESTORE

In [11], we provide a secure data store that offers operations to an authorized client application. We also provide an administration application that can call a method to add an application's credentials. This tool allows an administrator to add other applications with their certificates to the valid applications list. We sign the code for our administration application with the Mac App Store and hard code the certificate for the administration application into our service provider application. This hard coding enforces at installation time that the only application authorized to connect to the data store provider is the administration tool itself. Using the administration tool, we can grant access to the service for other applications. One of the services provided is the ability to add human user credentials that can

Algorithm 1. Process Authentication Algorithm.

3rd Party App Forks to Service App
Service gets parent pid
Service uses parent pid to get parent path
Service gest parent cert
Cert validated against valid apps

 ISBN: 978-1-61208-582-1

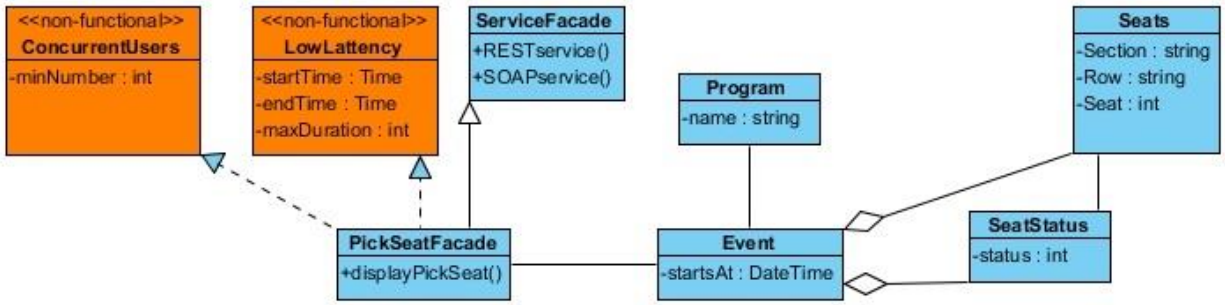

Figure 5. UML with Multi-Inheritance.

authenticate to the application. The addition of these application credentials allows data to be secured so only a specific application can get access or so a specific human user can gain access or by the combination of both the application and a user authenticated in the application.

VII. SECURE MODELING

The focus of our previous research work in secure modeling is to investigate the problem of modeling vulnerable partitions of a software application in the design and analysis phase of the software development lifecycle. We focus on two key areas: providing partition tolerance in cloud-based applications while maintaining application data integrity [14] and modeling non-functional requirements using standard UML design tools. Our contribution in this research field is to experiment with not only modeling application domain specific NFRs that are used in enterprise application architectures but also mapping the mode to application code that will enforce the requirement. Our hypothesis was that we could use standard modeling tools, traditionally used to model functional requirements, and extend them to allow modeling and code generation of NFRs. We modeled the NFRs using the extensibility mechanisms built into the standard modeling notations of UML and OCL to specify those NFRs. The models are exported to the XML standard XMI to enable our tooling to read the model. Java code is then generated to enforce each NFR by parsing the XMI of the model, matching the stereotype or OCL constraint to a Java fragment and producing the code.

In the first iteration of our work, different scalar values were represented by different stereotypes. For example, to represent different quantities of concurrent users, we had to create different stereotypes to represent each specific quantity, such as “1000 concurrent users” and “500 concurrent users”. Though this method allowed us the granular control to specify specific NFR requirements, there are two flaws. First, there is no way to group stereotypes into categories in standard UML notation. The stereotype only method does not allow any semantic relationship between the two stereotypes that both represent quantities of concurrent users. A developer would need to know the relationship to avoid making an error when switching between values, thus causing the semantics of the NFR to change completely. Second, on a large enterprise development project, the number of stereotypes required to represent all the different combinations of NFRs and scalar measurements would become unwieldy.

The solution we developed to solve both of these two challenges combines OCL and mock objects in the UML class diagrams. Specifically, we insert mock objects that provide new attributes to represent the scalar values measured in the enforcement of the NFR into the UML class diagram inheritance tree. The mock objects are generalizations that specify attributes that are inherited by the real façade objects. Once the new attributes are added through inheritance, we can specify standard OCL constraints to express the NFR and the appropriate measurement. Figure 5 shows a design using the mock multi-inheritance to enforce two non-functional requirements (“Low Latency” and “Concurrent Users”). Java code is generated from the mock objects using the single inheritance the programming language supports.

VIII. PLATFORM, EFFORT, AND SECURITY

With the advent of cloud computing, Platform-As-A-Service (PAAS) has become a way that a developer can leverage pre-built components to reduce the time to market. The goal of PAAS is to allow the developer to focus on the development of a solution for the business functions and not software functions that span many application domains. A good example of PAAS is force.com where the developer is provided many of the essential parts of an application out of the box. In [15], we evaluate the programming effort savings from leveraging different PAAS providers. In [16] we investigate the technical debt arising from software engineers ignoring the security vulnerabilities while developing software. In both works, we leverage COCOMO II [17] to estimate the development costs to track code leverage and technical debt accrued.

The 21st century has been dominated by bytecode compiled languages that have runtime engines that execute the code on different hardware platforms. The Java Runtime Engine (JRE) and the Microsoft .NET Runtime Engine (.NET) are the most dominant examples of the bytecode engines that free the developer from thinking about the underlying hardware. PAAS is the next evolution in freeing

 ISBN: 978-1-61208-582-1

up the developer's time so they can focus on the problem they are trying to solve instead of the technical plumbing required for the solution.

A hypervisor is computer software, firmware, or hardware, which executes virtual machines. A computer on which a hypervisor is called a host machine and each virtual machine is called a guest machine. Type 1 hypervisors run directly on top of hardware. Type 2 is a hypervisor that operates as an application on top of an existing operating system. If you were deploying an application to a Java PAAS today, it would be in a JRE running on a Type 1 hypervisor. OSv [18] is a JRE that can execute directly on a Type 2 hypervisor. Not having an extra operating system layer removes all the security vulnerabilities found in the OS layer below the JRE. Developing a solution that executes in OSv will be naturally more secure than other PAAS providers due to the fewer layers of potential exploits.

IX. CONCLUSION AND FUTURE WORK

In this paper, we described the work done in our lab to provide the missing modeling components, development tools, application architectures, and algorithms to increase the security guaranteed in software and improve the estimation of the effort in the SDL. Our current solutions are examples which prove that robust commercial solutions can be developed. Our future work includes developing a model-driven development solution that can be deployed on a secure bytecode runtime engine. The runtime engine should be capable of running directly on a hypervisor without the insecure extra layer of a traditional operating system.

REFERENCES

[1] Microsoft, Inc., "What is the Security Development Lifecycle ?," 2017. [Online]. Available: https://www.microsoft.com/en-us/sdl/. [Accessed 26 March 2017].

[2] B. Edmunds, Securing PHP Applications, New York: Apress, 2016.

[3] J. Walden, M. Doyle, R. Lenhof and J. Murray, "Java vs. PHP: Security Implications of Language Choice for Web Applications," in *Conference: Engineering Secure Software and Systems, Second International Symposium*, Pisa, Italy, 2010.

[4] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, pp. 51-59, 2002.

[5] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, pp. 37-42, 2012.

[6] A. Olmsted and C. Farkas, "Buddy System: Available, Consistent, Durable Web Service Transactions.," *Journal of Internet Technology and Secured Transactions (JITST)*, vol. 2, no. 1/2, pp. 131-140, 2013.

[7] A. Olmsted, "Heterogeneous System Integration Data Integration Guarantees," *Journal of Computational Methods in Science and Engineering (JCMSE)*, vol. 17, no. 51, pp. S85-S94, 2017.

[8] Tessitura Network. Inc, "Tessitura Software," 2017. [Online]. Available: http://www.tessituranetwork.com/en/Products/Software.aspx. [Accessed 13 August 2017].

[9] Zoho Corporation Pvt. Ltd., "Zoho CRM API," 2017. [Online]. Available: https://www.zoho.com/crm/help/api/. [Accessed 13 August 2017].

[10] S. C. Corp., Oracle9iDS Forms II: Customize Internet Apps Vol B, Sideris Courseware Corp., 2003.

[11] A. Olmsted, "Native Autonomous Process Authentication," in *Proceedings of World Congress on Internet Security 2016 (World-CIS 2016)*, London, UK, 2016.

[12] Apple Inc, "The Mac App Store," Apple Inc, [Online]. Available: http://www.apple.com/osx/apps/app-store/. [Accessed 01 June 2016].

[13] OS X Daily, "How to Show & Verify Code Signatures for Apps in Mac OS X," OS X Daily, [Online]. Available: http://osxdaily.com/2016/03/14/verify-code-sign-apps-mac-os-x/. [Accessed 01 June 2016].

[14] A. Olmsted, "Modeling Cloud Applications for Partition Contingency," in *Proceedings of the 11th International Conference for Internet Technology and Secured Transactions (ICITST-2016)*, Barcelona, Spain, 2016.

[15] A. Olmsted and K. Fulford, "Platform-As-A-Service Application Effort Estimation," in *Proceedings of The Eighth International Conference on Cloud Computing, GRIDs, and Virtualization (Cloud Computing 2017)*, Athens, GR, 2017.

[16] C. Brill and A. Olmsted, "Security and Software Engineering: Analyzing Effort and Cost," in *Proceedings of the The Third International Conference on Advances and Trends in Software Engineering* , Venice, IT, 2017.

[17] R. Madachy, "COCOMO II - Constructive Cost Model," [Online]. Available: http://csse.usc.edu/tools/COCOMOII.php. [Accessed 10 02 2017].

[18] Cloudius Systems, "OSv Designed for the Cloud," 2016. [Online]. Available: http://osv.io/. [Accessed 25 March 2017].

 ISBN: 978-1-61208-582-1