Automated Software Architecture Security Risk Analysis using Formalized Signatures

Mohamed Almorsy, John Grundy, and Amani S. Ibrahim
Centre for Computing and Engineering Software Systems
Swinburne University of Technology
Melbourne, Australia
[malmorsy, jgrundy, aibrahim]@swin.edu.au

***Abstract*— Reviewing software system architecture to pinpoint potential security flaws before proceeding with system development is a critical milestone in secure software development lifecycles. This includes identifying possible attacks or threat scenarios that target the system and may result in breaching of system security. Additionally we may also assess the strength of the system and its security architecture using well-known security metrics such as system attack surface, Compartmentalization, least-privilege, *etc*. However, existing efforts are limited to specific, predefined security properties or scenarios that are checked either manually or using limited toolsets. We introduce a new approach to support architecture security analysis using security scenarios and metrics. Our approach is based on formalizing attack scenarios and security metrics signature specification using the Object Constraint Language (OCL). Using formal signatures we analyse a target system to locate signature matches (for attack scenarios), or to take measurements (for security metrics). New scenarios and metrics can be incorporated and calculated provided that a formal signature can be specified. Our approach supports defining security metrics and scenarios at architecture, design, and code levels. We have developed a prototype software system architecture security analysis tool. To the best of our knowledge this is the first extensible architecture security risk analysis tool that supports both metric-based and scenario-based architecture security analysis. We have validated our approach by using it to capture and evaluate signatures from the NIST security principals and attack scenarios defined in the CAPEC database.**

***Index Terms*—Software security, Architecture Security Risk analysis, Formal attack patterns specification, Common attack patterns enumeration and classification (CAPEC)**

I. Introduction

Software architecture plays a vital role in the soundness and flexibility of complex software systems. While software architecture is usually expensive to change after system development, it is potentially cheaper to analyze early during system development [1]. This both helps in assuring that stakeholders' requirements have been met and aids in discovering flaws while modification is still a fraction of time and cost compared with later updates [2].

Architecture analysis has different goals. This includes assessing system maintainability, usability, sustainability, and security and resilience against attacks. Existing efforts to assess and evaluate software architecture against these quality attributes are classified into two main techniques: (i) scenario-based architectural analysis [1]-[3], focusing on generating (sometimes using brainstorming workshops) a set of evaluation scenarios based on the evaluation requirements; and (ii) metrics-based approaches [4]-[5], focusing on developing metrics that can be used in assessing software architecture.

Evaluating the security properties of software at early development stages helps in identifying security risks, potential security-related weaknesses in the software architecture, and areas that violate security requirements of stakeholders. These architecture and design flaws represent 50% of total reported vulnerabilities [6]. Many of these flaws cannot yet be discovered using existing security analysis tools. The security analysis task is usually conducted at different phases of the software development lifecycle under different names and using different artifacts [7]. Architecture security risk analysis is usually conducted at design phase using system architecture and design models. It targets identifying architecture and design security *flaws*. Vulnerability analysis is usually applied during development and testing using source code, or after system development using system binaries. These efforts target identifying existing security *bugs* in the system under test.

In this paper we focus on architecture security risk analysis. Most existing architecture security risk analysis efforts depend on a set of predefined metrics that have been hardcoded or implemented in the analysis tools [4, 8, 9]. Scenario-based efforts usually use security requirements and objectives as a source to develop the required security scenarios to be validated in a given software architecture [3, 10, 11]. Key problems are the lack of automated tool support for analyzing system architectures; lack of flexible and familiar architecture evaluation criteria specification language; limited consideration of the software operational environment capabilities' details.

To address these issues we introduce a new, comprehensive *architecture security analysis schema*. This schema captures details of a given system attack scenario including categories, preconditions, consequences, signatures, etc. A key entry is the *attack signature*. This signature specifies a set of invariants that, when matched, indicate that the given architecture vulnerable to the specified attack. We adopt the declarative and formal Object Constraint Language (OCL) [12] to capture such signatures. This makes it easier for a development team (usually familiar with OCL) to develop their own scenarios for assessing their software systems' architectures. We also use OCL to specify architectural security metrics used in assessing

system and security architecture soundness. Our approach supports extensible security metrics specification where new, user-defined, architectural security metrics can be introduced and evaluated without tool customization or development of new plug-ins. We have developed a system-security meta-model that helps in validating the OCL-based scenarios' and metrics' signatures. This meta-model is extensible, enabling users to capture other perspectives relevant to their architecture analysis task. Each attack or metric can be assigned a specific weight. This helps in performing automated architectural trade-off analysis between system potential architectures or between different systems. We capture system description and security in two different models to help in assessing security and architecture both separately and combined. The details of the system to be analyzed are captured in a system description model (SDM) using UML, with a UML profile that helps capturing interrelations between different system structure elements. Security details are captured in a separate security specification model (SSM). This captures security objectives, refined security requirements, security architecture (security zones, mechanisms, and services), and the security controls, patterns and functions used to realize the specified security.

Section II introduces a set of security scenarios and metrics we have identified from existing architecture security analysis efforts and we discuss possible signatures for these. Section III discusses our approach covering the attack scenario schema, signature specification, and our OCL-based analysis tool. Section IV discusses implementation details of our approach and Section V summarizes our evaluation results. Section VI discusses key strengths and weaknesses, and areas for further research. Section VII reviews the key related efforts.

II. Architectural Security Scenarios And Metrics

We discuss some of the security attack scenarios and metrics commonly used in assessing software architecture during the security risk analysis task. This is neither a comprehensive nor a complete list of possible scenarios or metrics. However, we try and cover most well-known scenarios and metrics frequently used. The example signatures used here are not meant to be complete or sound. Security experts have to develop detailed signatures that can be reused by other software engineers in assessing different systems.

A. Architecture Security Analysis Scenarios

Developing security scenarios to be used in assessing software architecture is a key task in scenario-based architecture analysis approaches. However, it is a very complicated task because it requires deep knowledge of the security domain, which is usually not feasible for all software engineers. The STRIDE model and EOP Card Game [13] give guidance in identifying such security scenarios. However, they still depend heavily on engineers' experience to analyze the architecture of the software under test. Recently, a new community effort introduced the Common Attack Pattern Enumeration and Classification (CAPEC) [1] as a reference that can be used in assessing systems' security. It provides a comprehensive list of possible attack patterns that are frequently used to breach systems' security. However, CAPEC is not yet formalized enough for use in automated architecture security analysis tools. We discuss below a few of the key patterns in this repository. We note that these attacks may have other signatures and specifications when it comes to source code level analysis (bugs) - i.e. for vulnerability analysis.

Man-In-The-Middle Attack: This attack intercepts communications between two components. The attacker makes independent connections with the victims and relays messages between them, making them believe that they are talking directly to each other. The signature of such attack is to have an unsecure connection between two components, or if the components communicate in an untrusted zone.

Denial-Of-Service (DOS) Attack: This attack aims to make a system or one of its key resources unavailable for legitimate users. DOS attacks have different formats with different signatures. Some use invalid inputs (in terms of type, format. Or size). Others overwhelm a system with requests. Possible signatures of such attack include: *(i)* a publicly accessible component that does not use input validation control (or firewall) to validate incoming requests, or *(ii)* a public interface that does not implement appropriate authentication control to filter requests.

Data Tampering Attack: An attacker can tamper with data at rest (storage), in transmission, or during processing if data is manipulated as plaintext. Possible signatures of these attacks include: *(i)* a system component that operates in an untrusted host (malicious insider), *(ii)* sending data between components or to a client in plaintext, or *(iii)* absence of appropriate security authorization control.

Injection Attack: This attack exploits the lack of input validation controls to pass in malicious inputs that can be used to gain higher privileges, modify data, or crash a system. Different types of injection attacks include SQL Injection, OS Command Injection, and XML Injection. The signature is that system components do not apply suitable input filtration on user inputs or on inputs from other untrusted components.

B. Architecture Security Metrics

Developing security metrics to be used in assessing software architecture is also a very complicated task. Different security metrics exist with different scope of applicability. These include: static vulnerability analysis metrics, dynamic vulnerability analysis metrics, static architecture security metrics, and runtime security metrics. We discuss some well-known metrics used in assessing architecture security.

1) System Architecture Security Metrics: These metrics help assessing the soundness of the software architecture security. Examples include attack surface metric [14], total public classified attributes and methods, critical super-classes proportion, least privilege, and least common mechanisms [8]. These metrics can be used to assess the exposure, exploitability, and attack-ability of the software system given its architecture, design, and may be code details. New architectural patterns such as multi-tenancy require new

[1] http://capec.mitre.org

security metrics that can assess tenants' data isolation, security elasticity, etc. Below we discuss examples of such metrics.

Attack Surface Metric [14]**:** This metric measures the proportion of the system that an attacker can use to attack the system. This can be measured as the number of system methods that receive data from the software environment, number of methods that return data to the software environment, number of communication channels, and number of untrusted data items. The larger the attack surface value, the more potentially insecure or vulnerable is the system.

Compartmentalization Metric: Compartmentalization means that systems and their components run in different compartments, isolated from each other. Thus a compromise of any of them does not impact the others. This metric can be measured as the number of independent components that do not trust each other (performs authentication and authorization for requests/calls coming from other system components) that the system is based on to deliver its function. The higher the compartmentalization value, the more secure the system.

Least Privilege Metric: This metric states that each component and user should be granted the minimal privileges required to complete their tasks. This metric can be assessed from two perspectives: from the security controls perspective we can review users' granted privileges. From the architectural analysis perspective this can be assessed as how the system is broken down to minimal possible actions i.e. the number of components that can access critical data. The smaller the value, the more secure the system.

Fail Securely Metric: The system does not disclose any data that should not be disclosed ordinarily at system failure. This includes system data as well as data about the system in case of exceptions. This metric can be evaluated from the security control responses – i.e. how the control behaves in case it failed to operate. From the system architecture perspective, we can assess it as the number of critical attributes and methods that can be accessed in a given component. The smaller the metric value, the likely more secure the system in case of failure.

2) Security Architecture Metrics: These metrics help assessing the soundness of the system security architecture and mechanisms including: security functions and components, security patterns, and security controls. NIST [15] introduces a set of design principles that should be adopted in developing secure systems. These include: use layered security; simplicity of the security design; protect information while it is being processed, in transit, and in storage; and never trust external inputs. We discuss a few examples that can be used to judge such characteristics.

Defense-In-Depth (Layered Security) Metric: This metric verifies that security controls are used at different points in the system chain including network security, host security, and application security. Components that have critical data should employ security controls in the network, host, and component layers. To assess this metric we need to capture system architecture and deployment models as well as the security architecture model. Then we can calculate the ratio of components with critical data that apply the layered security principle compared to number of critical components.

Isolation Metric: This assesses the level of security isolation between system components. This means getting privileges to a component does not imply accessibility of other co-located components. This metric can be assessed using system architecture and deployment models. Components marked as confidential should not be hosted with non-confidential (public) components. Methods that are not marked as confidential should not have access to confidential attributes or methods.

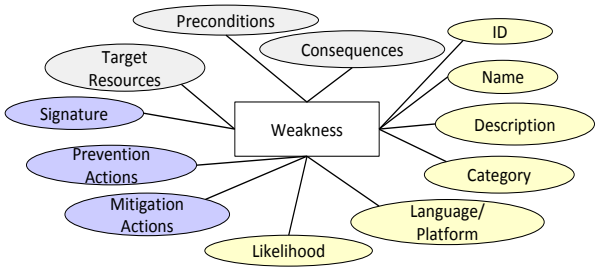

Fig. 1. Weaknesses Definition Schema

III. Our Approach

In our previous work [16], we introduced an OCL-based static vulnerability analysis approach supported with a toolset. This was based on capturing software vulnerability signatures as OCL invariants. These expressions are used in conducting program analysis of program source code or binaries to identify matches to OCL-specified vulnerability signatures. Our approach succeeded in locating static vulnerabilities with high precision and accuracy rates.

We now extend this approach here with a step up in the abstraction level. Instead of looking for signatures in source code we look for signatures (captured from security scenarios and metrics like those described above) in system architecture and design models. We integrate this with our original code vulnerability analysis approach. Our architectural security risk analysis approach is based on *(i)* a comprehensive security (attack) scenarios schema [16], shown in Fig 1, that captures details of a given scenario including relevant platform, likelihood, preconditions, consequences, etc.; *(ii)* a formal signature specification approach that can capture security scenarios and metrics signatures. Signatures are part of the attack scenarios schema; and *(iii)* an architecture security analysis tool that performs signature-based models analysis. Below we focus on the most interesting *signature* attribute.

A. Security Scenarios and Metrics' Signature Specification

Existing software security attack signatures in the Common Attack Patterns Enumeration and Classification (CAPEC) help understanding the nature of attacks. The same applies on existing security design principles and metrics. However, these are usually quite informally expressed and thus cannot be used in automatically locating potentials for such attacks in target systems. Applying them by hand is error-prone and time-consuming. Formalizing these descriptions allows architecture analysis tools to (semi)automate the analysis process. Ideally,

the formalization approach used should be extensible enough to support capturing new attacks' and metrics' signatures for different domains and requirements.

We use OCL as a well-known, extensible, and formal language to specify semantic signatures of security weaknesses and metrics. To support specifying and validating OCL-based signatures, we have developed a system-description meta-model, described in detail in [15], which captures system and security details from the high level objectives down to the source code entities and realization security controls. This model captures the main entities in an object-oriented system including components, deployment package, hosting services (web server), storage, communication channels, classes, instances, inputs, input sources, output, output targets, methods, new objects, objects interactions, etc. Moreover, it captures security concepts such as security objectives, requirements, architecture, zones, mechanisms, authentication, authorization, audit controls, etc. Each entity has a set of attributes, such as component name, provider, platform used, class name, accessibility, method name, accessibility, variable name, variable type, method call name, etc. This enables specifying of OCL-based scenarios' and metrics' signatures on different system entities with different abstraction levels.

Table I shows some attack scenarios' and simplified metrics' signatures specified in OCL using our system description model. These signatures can be further improved to incorporate system design details and even source code details, if available. These signatures should initially be developed by security experts and captured in a knowledge base, while software developers can further extend such signatures using customized and user-defined scenario and metric signatures.

B. OCL-based System Architecture Analyzer

After formalizing security scenarios and metrics' signatures in OCL an OCL-based analyzer component conducts static analysis of the system and security description details. This includes system source code represented in abstract syntax tee (optional); system design; architecture; and security models to locate and evaluate the specified security scenarios and metrics. Fig. 2. shows the architecture of our analysis component. To simplify the discussion of the analysis component architecture, we use example models from our test-bed Galactic ERP multi-tenant cloud application, a web-based ERP system [16]. Below we discuss the main inputs/outputs and components of our architecture security analysis tool.

System Description Model: Instead of using only the system architecture model to capture and apply security metrics, we use a detailed system description model – SDM. Fig. 3 shows the system description model of our exemplar Galactic ERP system [16]. The SDM is developed by system engineers using UML to describe details of the software. It describes system features, architecture, classes, behaviour, and deployment. These models cover most of the perspectives that may be required in analysing system architecture security soundness. Not all of these models are needed - it depends on system engineers and attack scenarios and metrics that they need to evaluate. Some system description details, such as class diagrams, can be reverse-engineered from source code.

TABLE I. EXAMPLES OF ARCHITECTURAL SECURITY SCENARIOS AND METRICS SIGNATURES IN OCL

ID	Metric Signature

1	`context System inv Man-in-the-Middle Attack:`
	`self.components->select(C1|`
	`C1.DeploymentZoneType = 'Untrusted'`
	`and self.components.exists(C2 |`
	`C2.Channels->exists(Ch |`
	`Ch.TargetComponent = C1`
	`and Ch.EncryptionControlDeployed = false)`
	`and C1.EncryptionControlDeployed = false`
	`and C2.EncryptionControlDeployed = false))`

	Any two components that communicate through an unencrypted channel and one or both of them operate in an untrusted zone or do not apply cryptography controls on their communicated messages.

2	`context System inv Denial-of-Service Attack:`
	`self.components->select(C1|`
	`C1.DeploymentZoneType = 'Untrusted'`
	`and C1.AuthenticationControlDeployed = false`
	`and (C1.InputSanitizationControlDeployed = false`
	`or C1.Host.Network.FirewallControlDeployed= false))`

	Any publicly accessible component that does not operate input sanitization control (or application firewall), and does not have authentication control.

3	`context System inv DataTampering:`
	`self.components->select(C1|`
	`C1.DeploymentZoneType = 'Untrusted'`
	`and self.components.exists(C2 |`
	`C2.Channels->exists(Ch |`
	`Ch.TargetComponent = C1`
	`and Ch.EncryptionControlDeployed = false)`
	`and C1.EncryptionControlDeployed = false`
	`and C2.EncryptionControlDeployed = false))`

	Any component that is deployed on an untrusted host (malicious insider) or zone, sends data in plain text, or does not operate authorization control.
4	`context System inv AttackSurface:` `self.components->select(C1| C1. DeploymentZoneType = 'Untrusted')->collect(C2 | C2.Functions)->size()`
	Number of functions defined in the provided interfaces of the public system components and number of functions defined in the required interfaces of the system public components that are used by other components.

5	`context System inv Compartmentalization:`
	`self.components->select(C |`
	`C.AuthenticationControlDeployed = true`
	`and C.AuthorizationControlDeployed = true)->size()`

	Number of architecture components that apply Authn. and Authz. controls on incoming calls (work independent and do not trust other components).

6	`context System inv FailSecurely:`
	`self.components->collect(C | C.Functions->select( F |`
	`F.IsCritical = true)->size())->sum()/`
	`self.components->collect(C |C.Functions->select( F |`
	`F.IsCritical = true)->size())->siz()`

	The average of critical methods and attributes in each system component.

7	`context System inv Defense-in-depth:`
	`self.select( C | C.IsCritical= true`
	`and C.AuthenticationControlDeployed = true`
	`and C.AuthorizationControlDeployed = true`
	`and C.CryptographyControlDeployed = true`
	`and C.Host.AuthenticationControlDeployed = true`
	`and C. Host.AuthorizationControlDeployed = true`
	`and C. Host.CryptographyControl = true)->size() /`
	`self.select( C | C.IsCritical = true)->size()`

	The ratio of critical components that have layered security compared to the total number of critical components in the system.

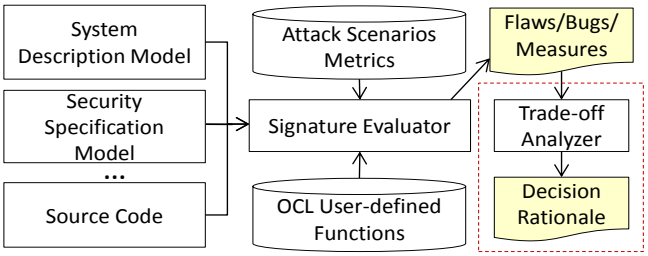

Fig. 2. OCL-based static security scenarios and metrics analysis tool

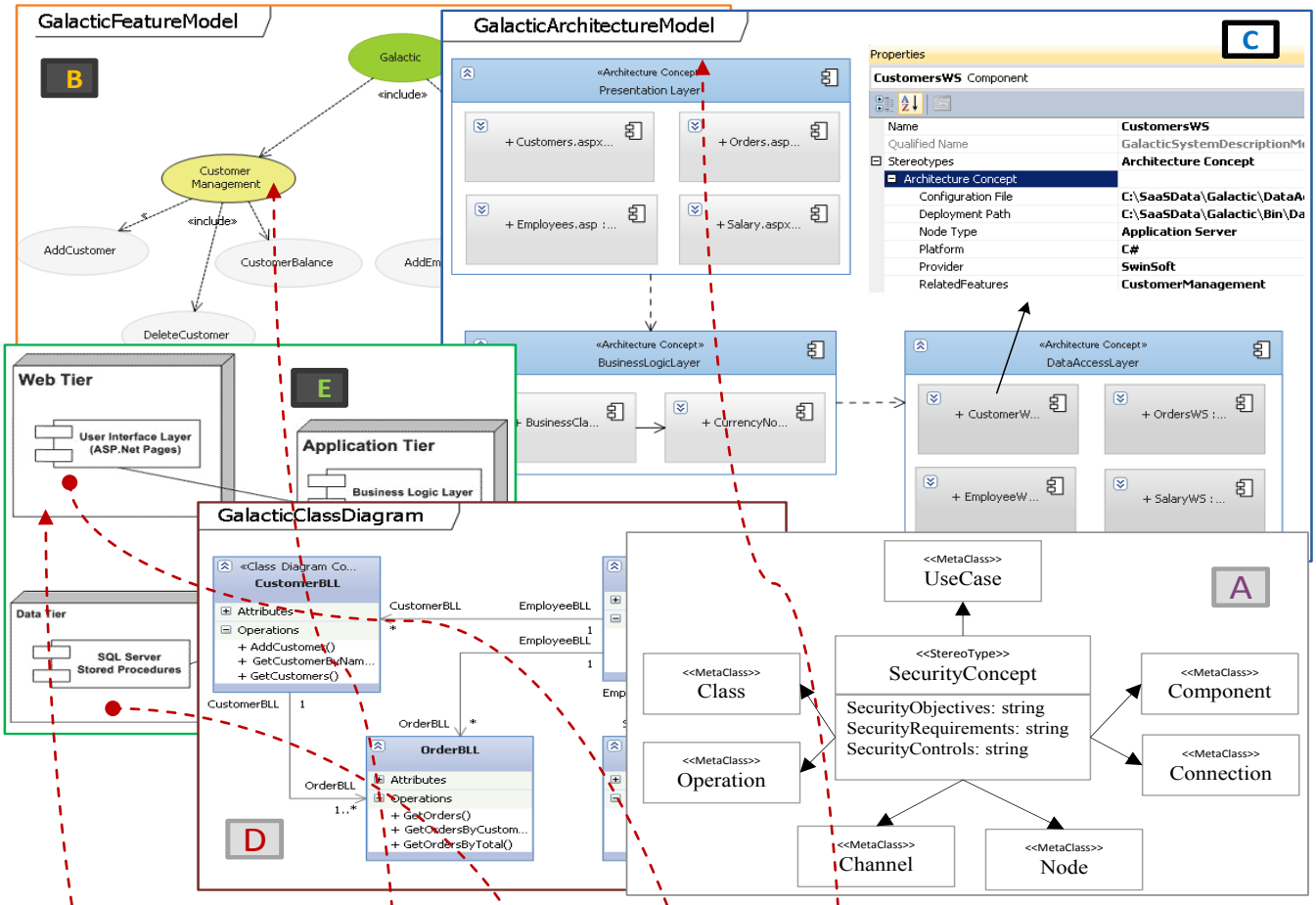

Fig. 3. Example of Galactic system description model

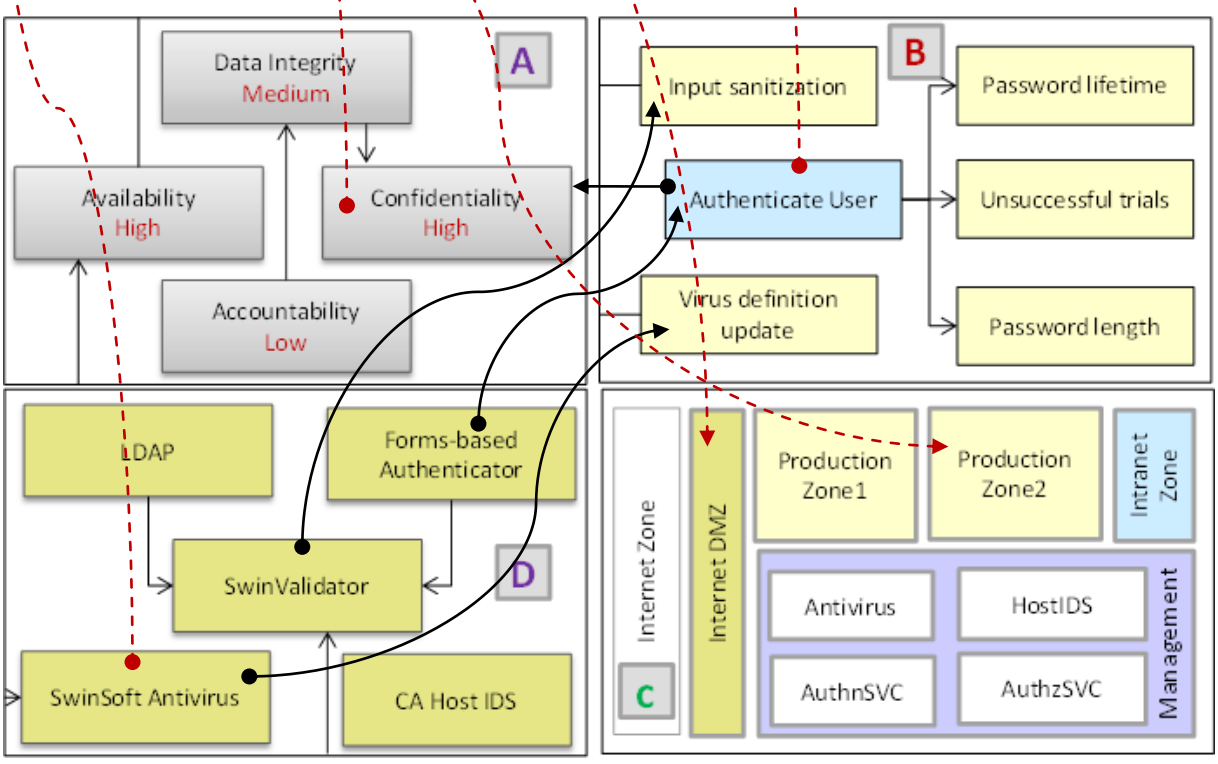

Fig. 4. Example of Galactic security specification model

To support mapping security specifications to system entities, we developed a new UML profile, shown in Fig. 3-A. This extends UML models with attributes that help in: *(i)* capturing relationships between different system entities in different models – e.g. a feature entity in a feature model with its related components in the component model and a component entity with its related classes in the class diagram; and *(ii)* capturing security entities (objectives, requirements, controls) mapped onto a system entity. It captures system features (Fig. 3-B) including customer, employee and order management features; system architecture including presentation, business and data access layers (Fig.3-C), system classes including CustomerBLL, OrderBLL, EmployeeBLL (Fig.3-D), and system deployment including web server, application server, and database server (Fig.3-E).

Security Specification Model: security engineers capture security details in a separate security specification model (SSM). This enables evaluating both system architecture details and security architecture details both separately and combined. We have developed a new, comprehensive security domain-specific visual language (SecDSVL). SecDSVL covers most of the details required during the security engineering process including: security goals and objectives, security risks and threats, security requirements, security architecture for the operational environment and security controls/patterns to be enforced. Here we just focus on objectives, requirements, architecture and controls. Not all these models are mandatory. Engineers decide which models they need to check or incorporate in their security analysis.

Fig.4 shows an example of the security specification model for the Galactic ERP system. This captures security objectives that should be satisfied (Fig.4-A), part of the security requirements (Fig.4-B), high level security architecture with security services and security mechanisms to be used in securing Galactic (Fig.4-C), and security controls and real implementations (Fig.4-D). The solid black lines between security entities reflect relationships between security entities – e.g. objectives and requirements, and requirements and realization controls/patterns.

System-Security Mappings: Engineers map security entities (objectives, requirements, controls) on system entities (features, components, classes). We support many-to-many mappings between security and system entities – i.e. many security entities could be mapped on many system entities. Mapping of security entities onto high-level system entities, e.g. a system feature, means that the same security entities are mapped to low-level system entities, e.g. components and classes. Moreover, mapping security objective (*O)* to a system entity (*E)* implies that all security requirements and controls that are linked to *(O)* are also mapped on *(E)*. The dashed red lines between Figures 3 and 4 show security to system mappings, such as placement of deployment nodes within security zones; security objectives that should be met on different components; and security solutions mapped to deployment node or system entities, etc.

Source Code Abstract Program Representation: to avoid being specific to programs written in a specific programming language or with a specific coding style, we transform the given system code into an abstract syntax tree (AST) representation. The program AST abstracts most of the source code details away from specific language constructs. We perform further abstraction of this AST using our system description model. This enables evaluating the conformance of source code with system and security models.

Signature Evaluator: This is the main component in our analysis tool. It receives the system and security details and security scenarios, vulnerabilities, and metrics to be evaluated, and generates a list of potential flaws, vulnerabilities, security holes, and security measures. During analysis, the signature evaluator loads the defined weaknesses and metrics in the signatures database (specified in OCL) and compiles these signatures into small analysis programs (using OCL_2_C# transformation that generates C# code from these signatures). These generated analysis programs analyze the fed in models to locate entities that match the specified signatures and calculate measurements specified. The user-defined OCL functions represent a repository of user-defined functions that can be used in developing complex scenarios and metrics signatures. This includes control flow analysis, data flow analysis, string analysis, taint-analysis, etc.

Trade-Off Analysis: The previous step produces a security analysis report with a list of security flaws and measurements. This report can be used to conduct trade-off analysis between different potential software architectures. The trade-off analysis component compares different architectures' metrics taking into account metrics weights. The output of is a recommendation on selected software architecture with rationale presented as a radar chart summarizing number of flaws and measurements between different systems or different system architectures, as shown in Fig. 5.

IV. IMPLEMENTATION

We briefly describe key implementation details of our formalized attack scenarios and metrics specification approach and supporting architectural risk analysis tool. We used Microsoft Visual Studio2010 UML modeler to capture system description models (as an SDM). We used Microsoft Visual Studio Modeling SDK to develop our SecDSVL, used in capturing security details, and our UML profile, used in mapping security details onto the target system SDM.

We developed a UI component using Visual Studio to assist system and security engineers in capturing security scenarios and metrics signatures' specified in OCL. This UI is based on our system description meta-model discussed in Section III. This checks the validity of OCL statements and tests specifications on simple target application models and source code. We use an existing OCL parser to parse and validate signatures against our system description model. Once validated, the signature is compiled into C# code that traverses system and security models to find matched flaws or to calculate security metrics' values. To parse the given program source code and generate a system abstract model, we use an existing .NET parser *NReFactory*, which supports VB.NET and C#. Moreover we have used a C parser written in python

called *pycparser*. We currently support locating attack patterns in C#, VB.NET, C/C++. We are working on parsers for PhP and Java. For a system without source code - i.e. only binaries are available - we use an existing de-compilation tool *ILSPY* to generate source code from binaries using .NET Languages.

V. EVALUATION

We performed a detailed evaluation to assess the capabilities of our approach in capturing signatures of software systems' architecture security evaluation criteria either as security scenarios or security metrics. Then we assessed its capabilities in identifying architecture flaws that match weakness scenarios and measure security metrics.

TABLE II. BENCHMARK APPLICATIONS SUMMARY

Benchmark	Downloads	KLOC	Files	Comps	Classes	Method
BlogEngine	>46,000	25.7	151	2	258	616
BugTracer	>500	10	19	2	298	223
Galactic	-	16.2	99	6	101	473
KOOBOO	>2,000	112	1178	13	7851	5083
NopCommerce	>10 Rel.	442	3781	8	5127	9110
SplendidCRM	>400	245	816	7	6177	6107

A. Benchmark Applications

We could not find a repository or benchmark set of software architectures to evaluate our approach and so we decided to use existing open source applications on which to conduct our experiments. We used reverse engineering to retrieve systems' architecture and performed manual analysis to identify security details from applications' source code. We have selected a set of six open source applications developed in .NET as our benchmark to evaluate our approach. Table II summarizes these applications including known number of downloads, size in lines-of-code, number of files, number of components, number of classes, number of methods. These cover a wide spectrum including: Galactic (ERP system developed for internal testing purposes); SplendidCRM (open source CRM); KOOBOO (open source Enterprise CMS for websites); BlogEngine (open source ASP.NET 4.0 blogging engine); BugTracer (open-source, web-based bug tracking); and NopCommerce (open-source eCommerce solution).

B. Evaluation Experiments Setup

To evaluate our benchmark applications' architecture security, we selected a set of four security attack scenarios (Man-in-The-Middle, Denial of Service, Data Tampering, and Injection attacks), and four security metrics (Attack Surface, Compartmentalization, Fail Securely, and Defense-in-Depth), some exemplar signatures and metrics are presented earlier.

We used a set of evaluation metrics to measure the soundness and completeness of our analysis technique. These metrics are precision rate, recall rate, and F-measure. The precision metric is used to assess the soundness of the approach. A high precision means that the approach returns more valid results (true positive - TP) than invalid results (false positive - FP). Thus the maximum precision is achieved when no false positives (Equation 1 below). The recall metric is used to assess the completeness. A high recall means the approach returns more valid results (true positive - TP) than misses valid results (false negative - FN), see Equation 2. The F-measure metric combines both precision and recall. We use it to measure the overall effectiveness (weighted harmonic mean). This metric depends on the importance of the recall rate and the precision rate e.g. if we are interested in high precision (more valid results) then we will give precision factor high weight, and vice-versa. In our evaluation, we assume that the importance of precision and recall is equal, see Equation 3.

$$Precision = \frac{TP}{TP+FP} \quad \text{Equation 1}$$

$$Recall = \frac{TP}{TP+FN} \quad \text{Equation 2}$$

$$F-Measure = 2\,\frac{Precision*Recall}{Precision+Recall} \quad \text{Equation 3}$$

These evaluation metrics can be applied directly on attack-scenario based approaches where we can count how many missed or invalid scenarios retrieved by our approach. However, most security metrics return values like average, min, max, etc. This means that we cannot apply our evaluation metrics directly on these security values – i.e. we cannot count how many system/security instances were missed or incorrectly selected. To overcome this, we have rewritten the metrics expressions (expand metrics' expression) into separate factors that we can examine (in terms of FPs, FNs).

C. Experimental Results

Except for Galactic, we did not have experience with these benchmark applications and their architecture, design, and security details. We used reverse engineering to retrieve parts of the system description models (mainly class diagram, sequence diagram and component diagram) from their source code repositories using Altova UModel. These benchmark applications were already developed with built in security functions. We performed manual analysis to identify security controls used in such systems (we use these details to develop systems' security specification models) and where they are currently applied (these details represent mappings between the security entities and system entities).

Table III summarizes the results of our experiments from our security scenarios and metrics analysis evaluation. Table III is divided into two parts: security scenarios, and security metrics. Columns represent IDs of the benchmark applications: (1) BlogEngine, (2) BugTracer, (3) Galactic, (4) KOOBOO, (5) NopeCommerce, (6) SplendidCRM. Rows represent flaws and metrics. We summarize for each application and each attack scenario or security metric analyzed the number of discovered flaws or the metric measured value; number of false positives (reported as flaw but the manual analysis showed it is not a flaw); and number of false negatives (a flaw, but missed by our tool).

From our experiments we found that our approach achieves on average (90%) precision over both security scenarios and security metrics, and on average (89%) recall rate on both. This means that in every reported (100) scenario instances our tool reports (90) valid scenarios and around (10) scenarios are missed. These values depend on the soundness of the scenarios and metrics' signatures.

TABLE III. EXPERIMENTAL RESULTS OF APPLYING OUR OCL-BASED ARCHITECTURAL SECURITY RISK ANALYSIS TOOL ON BENCHMARK APPLICATIONS. D: DISCOVERED FLAWS, M: METRIC MEASURED VALUE, FP: FALSE POSITIVES; AND FN: FALSE NEGATIVES

Scenario / Metric		[1]	[2]	[3]	[4]	[5]	[6]	Total
Security Scenarios								
Man-in-The-Middle (↓)	**D**	1	1	4	8	3	5	22
	FP	0	0	0	1	0	0	1
	FN	0	0	0	1	0	1	2
Denial of Service (↓)	**D**	1	1	3	2	1	2	10
	FP	0	0	0	0	0	1	1
	FN	0	0	0	1	1	0	2
Data Tampering (↓)	**D**	1	1	3	5	3	3	16
	FP	0	0	0	2	0	0	2
	FN	0	0	1	0	1	0	2
Injection Attack (↓)	**D**	2	1	3	5	4	3	18
	FP	0	0	1	1	0	1	3
	FN	0	1	1	1	0	0	3
Total	**D**	5	4	13	20	11	13	66
	FP	0	0	1	4	0	2	7
	FN	0	1	2	3	2	1	9
Average Precision = 90%, Recall = 87%, and F-Measure = 88%								
Security Metrics								
Attack Surface (↓)	**M**	8	11	17	23	18	24	101
	FP	1	2	2	1	2	4	12
	FN	0	0	1	3	2	1	7
Compartmental-ization (↑)	**M**	1	1	3	3	4	3	14
	FP	0	0	0	0	1	0	1
	FN	0	0	1	1	0	0	2
Fail Securely (↓)	**M**	0.3	0.2	0.5	0.5	0.4	0.6	-
	FP	2	1	0	0	0	1	4
	FN	1	0	0	0	1	1	3
Defence-in-Depth (↑)	**M**	0.5	0.5	0.8	0.4	0.3	0.5	-
	FP	0	1	0	0	1	0	2
	FN	0	2	0	1	0	1	4
Average Precision = 91% , Recall = 89%, and F-Measure = 90%								

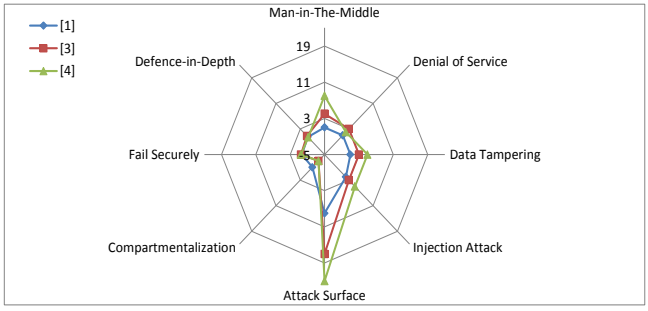

Fig 5. Example of the radar chart for applications 1,3, and 4

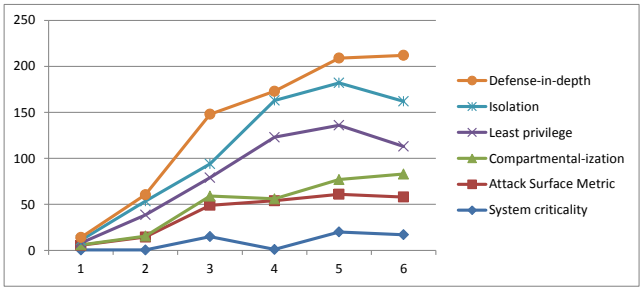

Fig 6. Performance of the analysis component

Table III also shows indicators associated with security metrics. If the indicator is (↑), it means that the higher the metric value, the more secure the architecture. The (↓) indicator means that the lower the metric value, the more secure the architecture. Totaling the security metrics has no sensible meaning as many have different units (some count, others use average or ratio). Table III shows that the man-in-the-middle attack is the most frequent vulnerability. We also have injection attack vulnerabilities including SQL Injection, OS Command Injection, XPath Injection. Denial-of-service was the least frequent injection attack vulnberability. When we compare these results with OWSAP TOP10 vulnerabilities, we found that they reflect relatively the same ranking where injection attacks are ranked number 1.

Although security metrics are helpful in comparing two different architectures for the same system (trade-off analysis), they are misleading as they depend on the application scale. Furthermore, in the security domain having just one flaw may result in breach of the whole system. Fig. 5 shows a radar chart of the attack scenarios and metrics reported for the applications in our benchmark. This chart assists in conducting trade-off analysis between different applications or different system architectures because it visualizes the different metrics' values for different application. Thus users can easily compare and select the best architecture from the security perspective. From this figure, one may decide to use application 1 instead of 4 (assuming both are in the same business domain) as it is more secure.

D. Performance Evaluation

Fig. 6 shows the time (in sec) required to analyze the benchmark applications' architectures to assessing specified security attack scenarios and metrics using the given set of scenarios and signatures shown in Table I. It is clear that the defense-in-depth metric takes much more time to identify than other metrics. The system criticality takes the lowest time. The time required to estimate a given security metric expression depends on the complexity of the specified OCL signature (transformed into C# code) and system size.

VI. DISCUSSION

To the best of our knowledge our approach is the first extensible architecture security risk analysis approach that supports both metric-based and scenario-based architecture security analysis. Using OCL provides a flexible, formal, familiar and extensible specification approach that can capture both metrics and scenarios signatures. These can be generic (applied on different systems and provide a knowledgebase), or application-specific (apply only to a specific application). A static scenario and metric analyser was developed based on our vulnerability signatures specification approach to perform analysis on system models at architecture, design and code levels. The scenarios and metrics database can be the responsibility of system engineers or even a community of security organizations to build up this repository. We have developed an architecture security analysis tool that can be extended without a need for new algorithms, modules, or

patches. Our current static analyzer achieves a precision rate of (90%) and recall rate (89%). These can be further improved using additional and more detailed signatures and more accurate description of a target system and its security details.
From our experiments we conclude that, in assessing application security, we cannot use measurements in percentages or ratios as they give misleading indicators as raw values. This is because results depend on system size. Moreover, we cannot use percentage metrics to assess different systems for the same reason. Although the number of found flaws is an important indicator, having one weak point as an attack surface means that the system can be attacked from this point. Some attack points are also much more vulnerable and likely to be exploited than others. These points can be measured using specialized metrics. We might then use an overall security metric for a target system [8] using weighted sum of the used measurements.

A key problem with our approach is that the results returned by the analysis tool depend on the soundness of the scenario and metric specifications (i.e. the OCL expressions). This can be mitigated by: (i) supporting a knowledge base with a set of covering metrics, allowing engineers to select metrics of interest; and (ii) developing a model-based security scenario and metrics-builder tool where engineers can build complex scenarios and metrics from existing small constructs – e.g. predefined scenarios and base measures.

Our security analysis tool works on XML representation of the software description model. This XML representation may be extracted from system architecture developed using UML, sysML, or user-defined domain-specific language. Moreover, we plan to try and automate the security attacks' scenarios and metrics' signatures from the existing attack' repository –e.g. CAPEC that violate customer security objectives.

VII. Related Work

Existing efforts in architecture analysis can be categorized into two main groups: scenario-based approaches and metrics-based approaches. Both have limitations related to approach formality in describing metrics or scenarios, extensibility to capture new metrics or scenarios to be assessed, and in automation of the architecture analysis process. A key notice from the existing efforts is that they focus mostly on scenario-based analysis. A possible justification of this tendency is that developing security metrics is a hard problem. Moreover, it limits capabilities of the approach compared to user-defined or tool-supported scenarios.

Scenario-based Analysis: Kazman *et al.* [17], Dobrica *et al.* [18], and Babar *et al* [19] introduce comprehensive software architecture analysis methods for different milestones. Kazman *et al.* introduce a set of criteria that can be used in developing or evaluating an architecture analysis method including identification of the goals, properties under examination, analysis support, and analysis outcomes. Babar *et al.* compare and contrast eight different existing architecture analysis approaches. A key weakness of all these approaches is a lack of tool support. Kazman *et al.* [20] introduce ATAM to identify trade-offs between quality attributes of a given system and report sensitivity points in its architecture. The approach is based on collaboration of stakeholders to define scenarios to evaluate architecture against. The analysis is expected to be manually. Faniyi *et al.* [21] extend the ATAM approach to support architecture analysis in unpredictable environments such as cloud computing platforms. They improve the scenario elicitation process using security testing with implied scenarios (unanticipated scenarios of components interactions). This generates potential scenarios that may lead to security attacks. Although this improved the scenario elicitation process, it still requires manual analysis. A further extension to our approach could be to integrate this approach as the source of our attack and metrics signatures. Halkidis *et al.* [11] introduce an architectural risk analysis approach based on locating the existing security patterns in the given system architecture using architecture annotation. Then, they use the STRIDE model to generate the set of possible attacks along with their likelihood. These security attacks can be mitigated using security patterns. Thus the lack of specific security patterns will cause violation of certain security objectives in the underlying system architecture. However, their approach does not support developing custom security scenarios to be analyzed in the target system. Admodisastro *et al.* [3] introduce a scenario-driven architectural analysis approach for black-box component based systems. Their analysis framework is extensible to support different pluggable analyzers that perform structure checking, quality checking, and conformance checking. However their proposed framework is high-level and lacks details of its components. Alkussayer *et al.* [2, 10] introduce a scenario-based security evaluation framework of software architecture. They use mapping of security goals/requirements, security patterns, and security threats to identify security scenarios used in evaluating (and improving) the given system architecture.

Metrics-based Analysis: Antonino *et al.* [4] introduce an indicator-based approach to evaluate architecture-level security in SOA. They use reverse engineering to extract security-relevant facts. They then use system-independent indicators and a knowledge base which maintains list of security goals and indicators relevant for every goal. Although the approach is extensible, it does not support automated analysis. Sant'anna *et al.* [9] describe a concern-driven quantitative framework for assessing architecture modularity. They introduce a set of modularity metrics that are used to assess a given system architecture. Alshammari *et al*. [8, 22] introduce a hierarchical security assessment model for object-oriented programs. They define a set of dependent metrics that capture security attributes of the given system. The proposed metrics are well organized. However, they are not extensible (i.e. are predefined metrics). Moreover, they do not consider security architecture details analysis. Heyman *et al.* [23] introduce an approach to identify security metrics to measure/assess based on mapping user security requirements on security objectives. For each security objective, they define security patterns that are expected to satisfy such objectives. Each security pattern has a set of security metrics that are satisfied by the pattern. The metrics specification approach is

informal so it does not enable automating the analysis phase. Sohr *et al*. [24] describe an architecture-centric security analysis approach. They reverse engineer system architecture from source code using the Bauhaus tool. They conduct manual analysis to identify security flaws existing in the given system architecture. Liu [25] introduce a service-oriented framework to analyze attack-ability of given software. They develop a new language to capture system architecture and security details. Using this model, they defined a set of built-in security metrics to be assesses in a given system architecture.

VIII. SUMMARY

We introduced a new architecture security analysis approach based on formalizing system architectural security attack scenarios and security metrics using OCL. Target system architecture and security details are captured using UML and our SecDSVL respectively. We have developed a prototype architecture security analysis tool that succeeds in analyzing different systems against different sets of security scenarios and metrics. We are able to apply these at source code, design and architecture levels. Our experiments show that security metrics should not be specified as ratio or percentage metrics as this gives misleading figures of a system's actual security.

REFERENCES

[1] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," In Proc. 1998 IEEE Int. Conf. on Engineering of Complex Computer Systems, pp. 68-78.

[2] A. Alkussayer and W. H. Allen, "Security risk analysis of software architecture based on AHP," In Proc. 7th Int. Conf. on Networked Computing, 2011, pp. 60-67.

[3] N. Admodisastro and G. Kotonya, "An architecture analysis approach for supporting black-box software development," In Proc. 5th European conference on Software architecture, Essen, Germany, 2011.

[4] P. Antonino, S. Duszynski, C. Jung, and M. Rudolph, "Indicator-based architecture-level security evaluation in a service-oriented environment," In Proc. 4th European Conference on Software Architecture, Copenhagen, Denmark, 2010.

[5] B. Tekinerdogan, "ASAAM: aspectual software architecture analysis method," In Proc. 4th Working IEEE/IFIP Conf. Software Architecture, 2004, pp.5-14.

[6] G. McGraw, *Software Security: Building Security In*: Addison-Wesley, 2006.

[7] W. D. Yu and K. Le, "Towards a Secure Software Development Lifecycle with SQUARE+R," In Proc. IEEE 36th Annual Computer Software and Applications Conf. Workshops, 2012, pp.565-570.

[8] B. Alshammari, C. Fidge, and D. Corney, "A Hierarchical Security Assessment Model for Object-Oriented Programs," In Proc. 11th International Conference on Quality Software, 2011, pp. 218-227.

[9] C. Sant'Anna, E. Figueiredo, A. Garcia, and C. J. P. Lucena, "On the Modularity Assessment of Software Architectures: Do my architectural concerns count?," In Proc. 6th Int. Workshop on Aspect-Oriented Software Development, Vancouver, Canada, 2007, pp. 183-192.

[10] A. Alkussayer and W. H. Allen, "A scenario-based framework for the security evaluation of software architecture," In Proc. 3rd IEEE Int. Conf. Computer Science and Information Technology, 2010, pp. 687-695.

[11] S. T. Halkidis, N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, "Architectural Risk Analysis of Software Systems Based on Security Patterns," In Proc. IEEE Transactions on Dependable and Secure Computing, vol. 5, pp. 129-142, 2008.

[12] M. V. Cengarle and A. Knapp, "OCL 1.4/5 vs. 2.0 Expressions Formal semantics and expressiveness," Software and Systems Modeling, vol. 3, pp. 9-30, 2004.

[13] T. Denning, T. Kohno, and A. Shostack, "Control-Alt-HackTM: A Card Game for Computer Security Outreach, Education, and Fun," 2012.

[14] P. K. Manadhata and J. M. Wing, "An Attack Surface Metric," IEEE Transactions on Software Engineering, vol. 37, pp. 371-386, 2011.

[15] G. Stoneburner, C. Hayden, and A. Feringa, "Engineering Principles for Information Technology Security (Baseline for Achieving Security), Revision A," NIST, 2004.

[16] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Supporting Automated Vulnerability Analysis using Formalized Vulnerability Signatures," In Proc. 27th IEEE/ACM Conf. on Automated Software Engineering, Essen, Germany, 2012, pp. 100-109.

[17] R. Kazman, L. Bass, M. Klein, T. Lattanze, and L. Northrop, "A Basis for Analyzing Software Architecture Analysis Methods," *Software Quality Journal,* vol. 13, pp. 329-355, 2005.

[18] L. Dobrica and E. Niemela, "A survey on software architecture analysis methods," *IEEE Transactions on Software Engineering,* vol. 28, pp. 638-653, 2002.

[19] M. A. Babar, L. Zhu, and R. Jeffery, "A framework for classifying and comparing software architecture evaluation methods," In Proc. 2004 Australian Software Engineering Conference, 2004, pp. 309-318.

[20] P. Clements, R. Kazman, and M. Klein, *Evaluating software architectures: methods and case studies*: Addison-Wesley Reading, 2002.

[21] F. Faniyi, R. Bahsoon, A. Evans, and R. Kazman, "Evaluating Security Properties of Architectures in Unpredictable Environments: A Case for Cloud," In Proc. of 9th Working IEEE/IFIP Conference on Software Architecture, 2011, pp. 127-136.

[22] B. Alshammari, C. Fidge, and D. Corney, "Security Metrics for Object-Oriented Class Designs," In Proc. 9th Int. Conf. on Quality Software, 2009, pp. 11-20.

[23] T. Heyman, R. Scandariato, C. Huygens, and W. Joosen, "Using Security Patterns to Combine Security Metrics," In Proc. 3rd Int. Conf. on Availability, Reliability and Security, 2008, pp. 1156-1163.

[24] K. Sohr and B. Berger, "Idea: towards architecture-centric security analysis of software," In Proc. 2nd Int. Conf. Engineering Secure Software and Systems, Italy, 2010.

[25] Y. Liu, I. Traore, and A. M. Hoole, "A Service-Oriented Framework for Quantitative Security Analysis of Software Architectures," In Proc. IEEE Asia-Pacific Services Computing Conference, 2008, pp. 1231-1238.