2015 International Conference on Engineering Complex Computing Systems (ICECCS 2015), Gold Coast, Australia, 9-12 December, (c) IEEE.

Improving Tenants' Trust In SaaS Applications Using Dynamic Security Monitors

Mohamed Almorsy Abdelrazek[*], John Grundy[#] and Amani S. Ibrahim[#]

[*] School of Information Technology, Faculty of Science, Engineering and Built Environment, Deakin Univ, Geelong, Australia

[#] School of Software and Electrical Engineering, Faculty of Science, Engineering and Technology, Swinburne University of Technology, Hawthorn, Victoria, Australia

mohamed.abdelrazek@deakin.edu.au, jgrundy@swin.edu.au, aibrahim@swin.edu.au

***Abstract*-It is almost impossible to prove that a given software system achieves an absolute security level. This becomes more complicated when addressing multi-tenant cloud-based SaaS applications. Developing practical security properties and metrics to monitor, verify, and assess the behavior of such software systems is a feasible alternative to such problem. However, existing efforts focus either on verifying security properties or security metrics but not both. Moreover, they are either hard to adopt, in terms of usability, or require design-time preparation to support monitoring of such security metrics and properties which is not feasible for SaaS applications. In this paper, we introduce, to the best of our knowledge, the first unified monitoring platform that enables SaaS application tenants to specify, at run-time, security metrics and properties without design-time preparation and hence increases tenants' trust of their cloud-assets security. The platform automatically converts security metrics and properties specifications into security probes and integrates them with the target SaaS application at run-time. Probes-generated measurements are fed into an analysis component that verifies the specified properties and calculates security metrics' values using aggregation functions. This is then reported to SaaS tenants and cloud platform security engineers. We evaluated our platform expressiveness and usability, soundness, and performance overhead.**

Keywords--Security monitoring, security metrics, run-time verification, Cloud computing monitoring

I. INTRODUCTION

The outsourcing of enterprise IT systems for hosting on third-party platforms e.g. cloud computing, either using Software-as-a-Service (SaaS) or Infrastructure-as-a-Service (IaaS) service delivery models, improves resources availability and accessibility at a manageable cost [1]. On the other hand, it increases customers' concerns about the security of their cloud-outsourced assets. The loss-of-control is a key security problem in adopting the cloud computing model where tenants do not have control on their outsourced assets hosted on the cloud. Enabling cloud tenants to monitor the security status and behavior of their cloud-hosted assets is a key enabler to improve tenants' trust and confidence in the cloud computing model. Of course design-time security analysis techniques are helpful, but usually not sufficient especially with the cloud computing model where SaaS application tenants, and their security requirements, are not known until run-time. Monitoring SaaS applications' security (mainly shared application instance model) can help in verifying system security properties (usually expressed as security policies) [2-4], discovering and reporting any violations of security requirements [5, 6], adapting system behavior and structure based on current context [7], or through proactive or corrective actions [8]. Assessing software systems security is usually formulated as a set of: (i) security properties, constraints or policies to be checked or enforced - e.g. only authenticated users can access resource X; and (ii) security metrics to be collected for the sake of security management and improvement - e.g. the ratio of system unauthenticated requests. Any increase in unauthenticated requests rate may imply that the system is probably under attack.

Software security monitoring platforms are usually designed with three components: security property and/or metric specification language, software instrumentation/profiling component (observing system behavior externally or using instrumentation), and security measurements' analysis (and may be reaction) component. Existing security monitoring efforts focus either on security properties or security metrics to be assessed, but not both. Property-based security monitoring efforts, usually referred to as run-time verification, focus on monitoring system security behavior to detect violations in the defined security properties or policies for security, reliability, etc. This includes MOP [9], MaC, Land JPaX[10], LARVA [11]. These efforts depend on adoption of formal languages in specifying run-time system (or security) properties. A key problem with many is that they do not support parametric properties i.e. a property that depends on parameters to decide what traces to consider in the analysis phase. MOP, one of the most active research approach, has been extended to overcome such limitations [12]. Dynamic security enforcement also has relevant efforts such as Polymer [13] SPoX [14] and in-lined-security policy enforcement efforts [15]. These efforts focus on monitoring system behavior and intercepting calls to relevant actions to enforce user-defined policies. MOP has been also

extended with a run-time security policy enforcement extension [16]. Using such formal specification languages complicates the adoption of such efforts by either security experts or software engineers. Most existing efforts assume that the system is to be assessed by software providers who are expected to have experience with formal languages such as linear temporal logic (LTL) [17], regular expressions (RE), context free grammar (CFG), or event calculus (EC) [5, 18, 19].

Metrics-based security monitoring efforts focus on providing frameworks that help in conducting security metrics development and analysis processes [20-24]. Using such approaches usually requires deep involvement of security experts and also tends to be both time-consuming and error prone. A key problem with these metrics-related efforts is the lack of automation. This is because they depend on informal metrics' definitions. Thus, security experts should get involved in developing/customizing monitoring tools.

Analysing efforts in both metric-based & property-based security monitoring areas we concluded that the formality, familiarity, abstraction, and extensibility of the language used in developing security properties and metrics are key issues. These factors are very important when addressing security monitoring of cloud-based SaaS applications from cloud tenants perspective (small experience/knowledge of the application). Moreover, to the best of our knowledge, there is no approach that supports monitoring and analysing both security properties and metrics for neither traditional software systems nor SaaS applications. We have formulated these issues into three key research questions that we address in this research:

- What details do we need to capture to fully describe a given security property or metric?
- How can we define formal signatures of such security metrics or properties using a familiar and relatively easy to use specification language for SaaS application tenants?
- How can we effectively use such formal specifications in automating the security monitoring process?

In this paper, we introduce a new, unified SaaS security monitoring approach based on capturing different SaaS tenants' security metrics and properties (policies) to be monitored as object constraint language (OCL) expressions [25]. Security properties and metrics defined in OCL expressions are used in automating the next two main security monitoring tasks: generation of security probes to be deployed with the monitored system to collect necessary measurements/events/traces; and generation of the security analysis programs that will be used in analyzing reported measurements. The analysis (checking) could work asynchronously (offline analysis) or synchronously (online analysis). We have evaluated our approach in capturing different metrics and properties, collecting and analysing measures, and its performance overhead.

In section II we introduce our unified monitoring platform and provide a usage example. In Section III we describe key platform architecture and implementation details. In Section IV we present our evaluation results and in Section V we summarise key contributions.

II. UNIFIED SECURITY MONITORING

In our previous work we introduced a signature-based static security analysis approach [26, 27]. We used OCL to capture static security threat and vulnerability signatures. These signatures are automatically transformed into security analysis programs that checks program source code and identify matches to the input OCL signatures. We also introduced a run-time security engineering approach (MDSE@R) [28, 29] that helps in enforcing different cloud SaaS application tenants' security requirements at run-time without modifying the target service source code using aspect-oriented programming (AOP). In our security monitoring approach, described here, we extend our signature-based static vulnerability analysis approach to support defining and analyzing system behavior properties and metrics. We reuse our MDSE@R approach to deploy these security probes within target system entities at run-time using dynamic Aspect-oriented Programming - AOP.

The details of what to capture to fully describe a security property or metric are encapsulated in a comprehensive security property/metric definition schema. A key entry in this schema is the property/metric signature, which we write using OCL. Security monitors are externalized from the target application(s), such that the application and security monitors could be easily changed the need of customizations – i.e. outline monitoring instead of inline monitoring. Reported measurements are stamped with tenant ID. Thus, we can discriminate between measurements reported for different tenants in case of shared, multi-tenant SaaS applications. In the next subsections we introduce our security metric/property definition schema, metric/property specification language, and the realization platform.

A. Security Metric/Property Definition Schema

We analyzed the key attributes used in defining security metrics or properties in existing efforts, and developed a security metric definition schema covering all such attributes, as shown in Fig. 1. Our security metric schema contains:

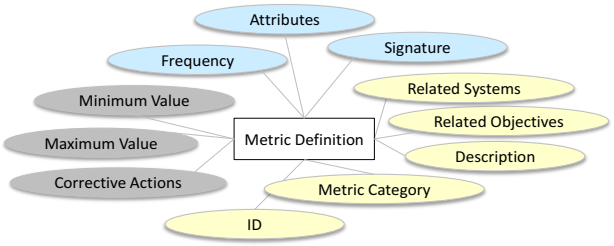


Fig. 1. Our security metric definition schema

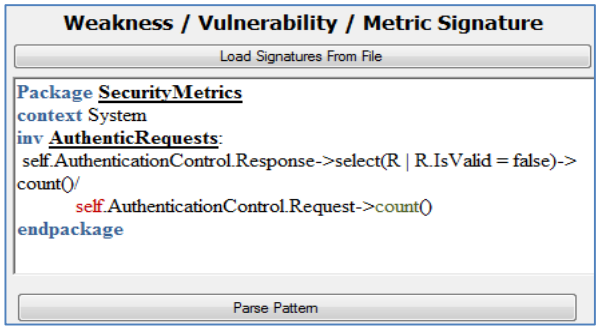


Fig. 2. OCL signature interface showing Authentic Requests Metric

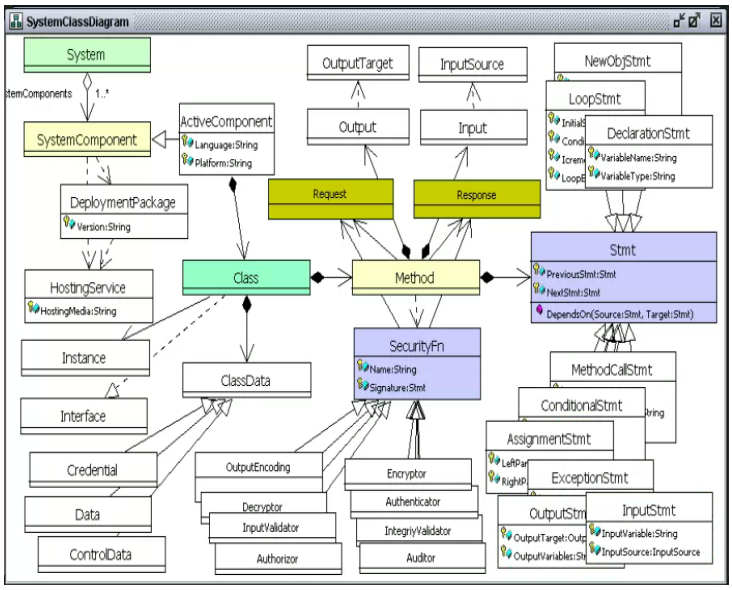


Fig. 3. A part of our system description meta-model

Property/metric ID: every security monitoring property or metric has a reference ID that identifies and links collected measurements. It also helps in uninstalling security probes whenever the metric/property is no longer needed. Metric ID includes tenant ID plus a string of alphanumeric characters. **Metric Category:** used in categorizing security metrics and properties into operational, management, static, dynamic metrics/properties for reporting purposes. **Related Objectives:** each security metric/ property could be linked to one or more of the tenant's security objectives that have been refined into a set of security metrics and properties. **Related Systems:** the target software system(s) in case we have multiple systems in the operational environment (hosted on the same platform). **Signature:** the security metric or property signature specifies the metric formula, property, rule, or expression to be evaluated, more details in the next section. **Attributes:** a list of relevant attributes to be enclosed in generated measurements e.g. target object, method name, arguments, return value, metric Id, timestamp, user identity. **Frequency:** each security metric definition includes the measurement frequency required. This may be every X-hours, X-days, X-weeks, X-months, and so on. The results collected by the security monitoring probes are grouped, evaluated, and consolidated (using metric signature) in a security status report to the security metric owner. **Minimum-Maximum Values:** each security metric has minimum and maximum values that define the valid boundaries of the security metric value. Whenever the security metric becomes below the minimum value or above the maximum value, security alerts should be reported to the security metric owner. **Corrective Actions:** a set of actions to be fired automatically when a violation is reported. In our previous work we showed how these actions are defined as changes to the software security model [30].

B. Security Metric/Property Signature Specification

We use Object Constraint Language (OCL) as a well-known, extensible, and semi-formal language to specify semantic security properties and metrics' signatures. To understand how OCL helps in developing security metrics and properties to be monitored we discuss few facts about OCL and compare it with other frequently used formal security property specification languages. First, OCL has four different collection types including set (no duplication of elements, no order), bag (duplicate, no order), ordered set (no duplicate, ordered), and sequence (duplicate, ordered). This is supported with a set of aggregation functions – e.g. maximum, minimum, average, variance, etc. that are frequently used in defining security metrics. OCL supports defining functions and procedures to be used in developing more complicated expressions. Moreover, it also supports declaring variables, and control statements (conditions and loops).

LTL and Event Calculus (EC) languages (two key languages in run-time verification domain) depend on defining a sequence of system events that the system should follow or that should not happen at all. To help OCL to support defining properties over a sequence of event, each generated measurement by our approach has a timestamp and all properties' related system measurements are maintained in ordered set based on reporting time. This helps in specifying sequences and time-related properties. The LTL language

depends on four main operators: always, until, next, and eventually. We can express similar operations in OCL using: for all, control statements, last, and exists operations. Event Calculus is based on events/actions and fluents (variables). EC defines a set of predicates that help in defining expected system behavior to be verified including: HoldsAt, Initiates, Terminates, and Happens operations. OCL supports defining such predicates that evaluate the occurrence of certain events as base expressions that can be reused in expressing and assessing complex expressions. We give examples of possible derived security metrics and compound predicates later.

However, OCL suffers from ambiguity problems. To support specifying valid OCL-based metrics/properties signatures, we have developed a system-description meta-model, as shown in Fig. 3. This model captures key concepts (semantics) in object-oriented systems including components, deployment package, hosting services (web server), classes, instances, inputs, input sources, output, output targets, methods, method body. We also use it to capture key security mechanisms such as: authentication, authorization, audit controls, etc. Each entity has a set of attributes, such as component name, provider, platform used, class name, method name, accessibility, variable name, variable type, method call, etc. This model captures static system attributes, and is used as a reference to develop signatures as discussed in Section IV.

To capture dynamic system attributes, we extended each system entity in this meta-model (application, components, classes, methods, properties, and security controls) with two concepts: *Request* – in run-time verification domain to capture before events; and *Response* – in run-time verification domain to capture after events. The details involved with such concepts change according to the system entity - i.e. it change from component to class down to method. Moreover, these attributes can be extended as needed. Similarly, the system description meta-model can be extended to include new concepts in metric definition and analysis.

Fig. 2 shows a snapshot from our tool where tenants specify their metrics' signatures. In this figure we show a signature of "authentic requests ratio" metric. This metric has been defined to measure the ratio of invalid authentication requests (authentication control response said that it is not valid) divided by the total number of requests received by the security authentication control. Any increase in this ratio reflects the possibility of being under attack to break the application-operated authentication control. Several example metric and property signatures are given in the Evaluation section below. The authentication control, and its attributes, is an abstract concept (model entity), software engineers will need to define the actual control – i.e. the actual security authorization function or API – in the application platform profile. A platform profile is an *optional* XML configuration file per service used to define a mapping between abstract model concepts that we cannot extract from code (if the code is available, such as security functions). This platform profile is used to define the actual system entities we need to intercept for security monitoring. The reason to introduce a platform configuration file is to that tenants do not know whether the system uses pre-defined security libraries (e.g. .Net Membership) or custom security libraries. Software engineers, because they know the system, can avoid this profile by defining special method calls (for a security control) as a basic metric and reuse the metric in composite metrics.

C. *Security Monitoring Platform*

After formalizing security metrics and properties signatures using OCL, we need to extract security probes, deploy them within the software, collect measurements generated by these probes, analyse these measures, and either trigger correction action or consolidate results into status report. The architecture of our security monitoring platform is shown in Fig. 4 as discussed below.

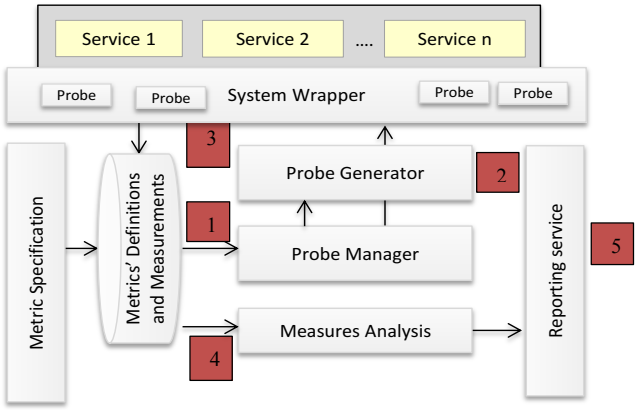


Fig. 4. Our security monitoring platform

Probe Manager (Step 1): once a security metric or property is added or updated in the repository, the security probe manager does the following: (i) Triggers the probe generator component to generate necessary security probes with corresponding attributes; (ii) Deploys the generated security probes within the system using the system wrapper component and remove deployed probes if the metric is no longer required; and (iii) Adds a new entry in the metrics analysis timer service according to the metric specified frequency. The analysis service analyzes new measurements and notifies the reporting service.

Probe Generator (Step 2): the main responsibility of the probe generator is to extract security probes from a given metric definition. The probe generator needs to extract: entities to monitor, and attributes to collect. The list of entities to be monitored is extracted from parsing the OCL abstract syntax tree (AST). This list is passed to the system wrapper to add interceptors to the system using AOP. The defined metric attributes are used to generate a measurement class that extracts actual values from the system wrapper (interceptor) at

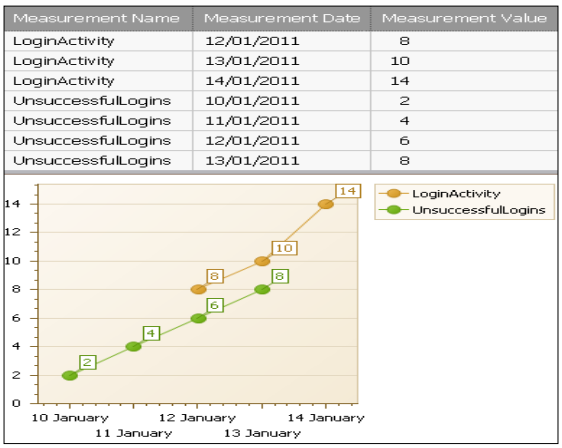
request/response interception time and send these measures to the monitoring platform – e.g. Given an OCL signature such as "self.AuthenitcationControl.Request…", this results in a security probe that intercepts requests to the system authentication control, Another example "…Method.Name='login'… ", we generate a probe to intercept requests to login method in a selected class. A probe for a method request will report measurement with: metric Id, tenant Id, timestamp, method name, arguments, and target object. A measurement for a method response will have: metric Id, tenant Id, timestamp, method name, arguments, target object, and return value.

System Wrapper (Step 3): the system wrapper is responsible for injecting interceptors (using dynamic aspect-oriented programming - AOP) within the target system/service at run-time at the critical points (system entities that have security properties or metrics defined on them). The system wrapper supports two modes of interception: *synchronous* where the system is intercepted and put on hold until the security analysis service confirmation (active monitoring tasks, such as intrusion prevention systems or application firewalls). This is usually used to handle security properties that should be verified at run-time; or *asynchronous* to reduce performance overhead (passive monitoring). We use this model with security metrics. The system wrapper that we have implemented currently supports intercepting requests at the system level, component level, and method level, which is adequate for the goal of security monitoring and analysis [31, 32].

Measures Analysis Service (Step 4): the analysis service parses the specified security property or metric signature developed in OCL and generates a C# analysis class. These security analysis classes deployed within the analysis service and loaded at run-time. Then, according to the metric frequency, these classes get executed. These security analysis classes check the measurements collected looking for violations. Fig. 6 shows an example of analysis class generated from the specified metric in Fig. 2. The code simply counts how many measurements collected for authentic requests metric where the authentication control return invalid authentication compared with the total number of requests. The output of the analysis service is sent to the repository for further use by other metrics (derived metrics, to discuss later), and for historical analysis.

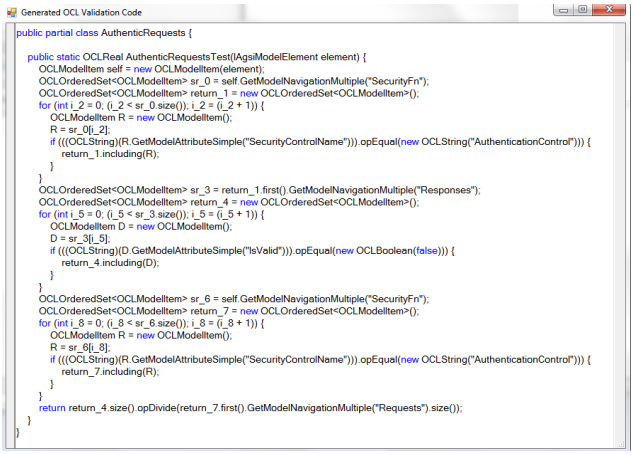
Reporting Service (Step 5): takes the aggregated results stored in the repository for the target system and tenant and provides a set of visualizations for tenant security engineers. Currently a set of tabular and chart visualizations are supported and accessed via a web page, as shown in Fig. 5. This figure shows the metrics' values for two different metrics and the trend of such metrics. This is helpfully to understand how and when the system behavior changes. Security experts will have to investigate in root causes of such changes. We are extending the reporting service with a visual designer to help designing how metrics will be visualized [33].

Tenants can define a set of mitigation or recovery actions to automatically apply in case of security property/metric violation. Currently, we support injection of security controls at run-time at the critical points (source of violations) through virtual patching using MDSE@R [30].



Measurement Name	Measurement Date	Measurement Value
LoginActivity	12/01/2011	8
LoginActivity	13/01/2011	10
LoginActivity	14/01/2011	14
UnsuccessfulLogins	10/01/2011	2
UnsuccessfulLogins	11/01/2011	4
UnsuccessfulLogins	12/01/2011	6
UnsuccessfulLogins	13/01/2011	8

Fig. 5. Example of our metrics analysis user interface



```
Generated OCL Validation Code
public partial class AuthenticRequests {

    public static OCLReal AuthenticRequestsTest(IAgsiModelElement element) {
        OCLModelItem self = new OCLModelItem(element);
        OCLOrderedSet<OCLModelItem> sr_0 = self.GetModelNavigationMultiple("SecurityFn");
        OCLOrderedSet<OCLModelItem> return_1 = new OCLOrderedSet<OCLModelItem>();
        for (int i_2 = 0; (i_2 < sr_0.size()); i_2 = (i_2 + 1)) {
            OCLModelItem R = new OCLModelItem();
            R = sr_0[i_2];
            if (((OCLString)(R.GetModelAttributeSimple("SecurityControlName"))).opEqual(new OCLString("AuthenticationControl"))) {
                return_1.including(R);
            }
        }
        OCLOrderedSet<OCLModelItem> sr_3 = return_1.first().GetModelNavigationMultiple("Responses");
        OCLOrderedSet<OCLModelItem> return_4 = new OCLOrderedSet<OCLModelItem>();
        for (int i_5 = 0; (i_5 < sr_3.size()); i_5 = (i_5 + 1)) {
            OCLModelItem D = new OCLModelItem();
            D = sr_3[i_5];
            if (((OCLString)(D.GetModelAttributeSimple("IsValid"))).opEqual(new OCLBoolean(false))) {
                return_4.including(D);
            }
        }
        OCLOrderedSet<OCLModelItem> sr_6 = self.GetModelNavigationMultiple("SecurityFn");
        OCLOrderedSet<OCLModelItem> return_7 = new OCLOrderedSet<OCLModelItem>();
        for (int i_8 = 0; (i_8 < sr_6.size()); i_8 = (i_8 + 1)) {
            OCLModelItem R = new OCLModelItem();
            R = sr_6[i_8];
            if (((OCLString)(R.GetModelAttributeSimple("SecurityControlName"))).opEqual(new OCLString("AuthenticationControl"))) {
                return_7.including(R);
            }
        }
        return return_4.size().opDivide(return_7.first().GetModelNavigationMultiple("Requests").size());
    }
}
```

Fig. 6. C# code generated from metric signature in Fig. 2

III. IMPLEMENTATION

We briefly describe key implementation details of our automated, unified security monitoring and analysis tool. *First,* we developed a UI component to assist security experts in developing their security metrics definitions and signatures using OCL. This provides security metric specification and signature editing including checking validity of OCL expressions and testing of specifications on sample measurements. We use an existing OCL parser [34] to parse and validate signatures against our system description model. Once validated, the metric definition is stored in the repository.

Second, we refined an existing OCL-to-C# translator library to transform the developed metrics' signatures into C# analysis classes. Each class has a single static method that accesses the

metrics' and measurement repository and applies the C# code on these measurements. An example of such generated C# code is shown in Fig. 6.

Third, we developed a probe generator library that analyses the OCL expressions and extracts the list of entities to monitor. We then generate a probe class that simply copies data from the system execution context to the class data members – e.g. copy the current user id to the UserID measurement attribute, copy method inputs to the measurement method input attributes, etc. The default measurement attributes (Metric ID, timestamp, etc.) are set with defaults. These results are sent back to the analysis service.

Fourth, we developed a system wrapper to help in injecting security probes at the critical system entities. To support run-time system interception, our platform combines both dependency injection and dynamic-weaving AOP approaches. The system wrapper supports wrapping of both new developments and existing systems. For new development, we use the Microsoft Unity application block delivered by Microsoft PnP team to support intercepting any arbitrary system entity. Unity supports dynamic run-time injection of interceptors on methods, attributes and class constructors using system configurations. For existing systems we adopted Yiihaw AOP tool to modify application binaries (dll and exe files) by adding aspects at any arbitrary system entity. In the latter case, we add a direct call to our system wrapper. The system wrapper is updated with deployed probes. At a given request for a given tenant, it triggers tenant metric/property probe object.

IV. EVALUATION

In this section we summarize our evaluation experiments we performed to assess the capabilities of our approach in defining, generating, deploying, collecting measures, and analyzing security metrics/properties details. We defined three key objectives to address in our evaluation: (i) Approach expressiveness by developing a range of security metrics and properties, as shown in Table 2, and comparing the expressiveness of our OCL-based specifications to other formalisms; (ii) Approach soundness applied on example multi-tenant SaaS application; and (iii) performance overhead.

A. Approach Expressiveness

The expressiveness and usability evaluation experiments of our specification language covered capturing definitions of security metrics and properties ranging from simple condition and counting of raw measures up to complicated security metrics and properties that incorporate results from other properties. To the best of our knowledge, there is no benchmark of security properties and metrics (we found a new project led by OWASP [35] to come up with critical security metrics in assessing web software systems, but it is still in the inception phase). To overcome this problem, we did two exercises: (i) a comparative analysis of our OCL-based approach against existing specification languages; and (ii) used our approach to define a set of security properties introduced in different research papers.

Comparative Analysis. We compared our approach against Event Calculus with different extensions as explained in [36], PMSL (performance metrics specification language) [37], and GMSL (Goanna Metric Specification Language) [38]. EC-Assertions [36], a first-order predicate calculus that is used to develop formal specifications and in reasoning about system properties that could be specified in terms of a set of events and their effects where the occurrence of these events impacts the satisfaction of these system properties. Performance Metric Specification Language (PMSL) [37, 39] was developed to capture high-level user-defined parallel-systems performance metrics. Metrics expressed in the PMSL are fed in the G-PM performance analysis tool. PMSL is a declarative functional language. PMSL does not include control flow or state altering constructs. The PMSL provides a couple of set operations and aggregation functions. The list of measurable objects to be monitored is limited to a predefined list. GMSL [38], the Goanna Metric Specification Language, was developed to assess program source code quality using a set of user-defined source code metrics assessed using model-checking.

TABLE 1 . Comparison of metric/property specification languguages

Criteria	**EC**	**OCL***	**PMSL**	**GMSL**
Applications	Reasoning / verification of sys. Props.	Assessing system properties & metrics	Parallel systems performance metrics	Static program quality metrics
Key Features	Events, properties, relations	Declarative, properties, relations & set fns	Declarative, Built-in attributes, metrics & fns.	Declarative, built-in code analysis functions.
Operators	✓*	✓	×	×
Quantifiers	✓*	✓	✓	✓
Temporal Events	✓*	✓	✓	×
Pre-cond	✓*	✓	✓	✓
Post-cond	×	✓	×	×
Complexity	×	✓	✓	✓
Extensibility	×	✓	×	×
Domain Specific	✓°	✓	●	●
Limitations	No aggregation or historical Fns	A bit lengthy compared to LTL.	No control flow, no alterable state	Work on codebase AST only

×: not supported ✓: supported ✓* supported as a language extension
● support one domain ✓° Require user involvement

Table 1 summarizes the comparison we did between these languages and our OCL-based approach. The criteria we used include: possible applications of the language, supported features, limitations, supporting specification of Boolean logic expressions, logic quantifiers, temporal events, pre-conditions and post-conditions, complexity of the language from the user perspective, is domain specific, and extensibility to user defined metrics/properties. We selected these criteria because they represent the key constructs in defining most of the properties/metrics. Table 1 illustrates that using these assessment criteria, our new OCL-based language is more rich and expressive in developing different types of metrics than the compared approaches. OCL can help in capturing system properties in design models [26] and source code level [40], and now dynamic properties and metrics. OCL is based on set theory. It supports development of boolean expressions using and/or operators. The original design of OCL targeted development of pre and post constraints/properties, easy to extend with new domain concepts as we did in our language using our meta-model, supports different aggregation functions, which are useful in metrics specification.

Specification of Security Metrics and Properties. In this experiment, we developed 10 security properties and metrics to assess the practical expressiveness of our approach. The security properties we used were collected from different research papers written in different formal languages: a set of security properties in [41] are modelled in Event Calculus, examples from [13] using Polymer language, and a set of policies introduced in [16] using MOP CFG. We were able to successfully specify these properties using our OCL-based approach, as shown in Table 2. The signature of these properties in other formal languages can be found in the sources above.

Information Disclosure Property [41]: The information communicated from agent A should not be disclosed to agent B unless it has been authorized – i.e. we must find a valid authorization record before proceeding with a system response accessing confidential information.

Chinese Wall Policy [16]: This policy states that subject S should not be able to access object O in the same conflict of interest datasets. This means that we cannot find a request to method M (read object O) of conflict in any of the methods of S.

Restrict System Calls [16]: Restricting the program from accessing system resources. This could be done by disabling execution of external code, such as OS system calls. In this policy signature we locate requests to the SystemHandler class (assumed to by the class responsible for external code execution) and simply return false if any happened. The authors in [16] have specified call bypass action. This could be achieved through our system wrapper.

Separation of Duties: Disallow a user to perform more than one action for a given request. The user might have privilege to execute a certain action, but they might be disallowed to perform other actions that contradict with actions they have performed. Thus we assume that the user is already authorized. We also assume that the focus is that the user cannot do all actions on one object – i.e. the user may do X operations on a given cheque but not all operations. An easier version could be to limit user to defined actions.

Authenticated Requests Metric: the ratio of requests received by the authentication component against total system requests. The higher this ratio, the more secure the system.

TABLE 2 . Example signatures of security metrics/properties in OCL

Metric	Signature
Information Disclosure	context Method inv InfoDisclosure: Let access : Request := self.Requests->last() in Let authorized : Response := self.AuthorizationControl.Responses-> select(R| R.IsValid = True AND access.UserID = R.UserID)->last() in IF (authorized) THEN true ENDIF
Chinese Wall	Let Subject := Classes->select(Name = 'Subj')->first() in Let Obj: Class := Classes->select(Name = 'Object')->first() Let mthdCall : Request := self.Requests->last() in Let mthdReturn: Response := self.Responses->last() in Let access : Request := self.Requests->last() in IF (access.RequestTime > mthdCall.RequestTime and access.RequestTime < mthdReturn.ResponseTime) THEN Not self.Conflictlist->exists(R| R = access.Target)
Restrict System Calls	Let SystemCalls : Request := Classes->select(Name = 'SystemHandler')->first().Requests()->last() in IF (SystemCalls <> null) THEN false ENDIF
Separation of Duties	Let xReq : Request:= Requests(Entity = 'MthdX') in Let yReq : Request:= >Requests(Entity = 'MthdY') in Let zReq : Request:= >Requests(Entity = 'MthdZ') in IF (xReq.UserID = yReq.UserID and xReq.Target = yReq.Target Or xReq.UserID = zReq.UserID and zReq.Target = zReq.Target Or yReq.UserID = zReq.UserID and xReq.Target = yReq.Target) THEN false ENDIF
Authenticated Requests	context System inv AuthenticatedRequests: self.AuthenticationControl.Requests->select()->count()/ self.Request->select()->count()
Authentic Requests	context System inv AuthenticRequests: self.AuthenticationControl.Response->select(R | R.IsValid = true)->count()/ self.AuthenticationControl.Request->select()->count()
Last(10) Authz. Reqs	context System inv Last10AuthzCtl: self.AuthorizationControl.Requests->select()->Last(10)
Top(10) admin Requests	context System inv Top10AuthnCtl: self.AuthenticationControl.Responses->select(R | R.UserID = 'Admin')->count()
Mean Time Between Unauthentic Request	context System inv MTBUnauthenticRequests: self.AuthenticationControl.Responses->select(R | R.IsValid = false)>differences('Measurementtime')-> sum() / self.AuthenticationControl.Responses->select(R | R.IsValid = false))->count()
Authenticated Requests Trend	context System inv Authenticated RequestsTrend: self.AuthenticatedRequests.Differences('AuthenticatedRequests')->sum() / self.AuthenticatedRequests-> count()
MTBUR Over Systems	context System inv MTBUROverSystems: self.MTBUnauthenticRequests->sum()/ self.MTBUnauthenticRequests->count()

Last (e.g. 10) Authorization Requests: This metric is used to take a random sample of the recent requests sent to the authorization security control. This metric can be used by admins to check the details (e.g. identity of requesters) of requests sent to the authorization security control after certain period of the day – e.g. out of the working hours.

Top (e.g. 10) admin Authentication Requests: This metric could be used by management to check how frequently administrators logged in to the system in the last period. Such metric can be detailed to reflect details of these requests including time of these requests, IP (source) of these requests, etc. It can assist identifying several vulnerabilities, including components with excessive privileges or lacking isolation.

Average Time between Unauthentic Requests: this metric measures the average time between consecutive unauthentic requests reported by the authentication control. A high measurement value means the underlying system is stable and secure. This metric could be used with authentic requests metric to know if the system is under attack or not.

Developing complex properties and metrics is always a requirement in any monitoring and analysis domain. These complicated metrics make use of other basic security properties/metrics. In Table 2, the last two rows show examples of complex, derived security metrics.

Security Metric Trend: This security metric helps in assessing the trend of certain metric values over a period of time. Here, we apply it on the authenticated requests metric defined above. This helps in figuring out whether there is an increasing or decreasing trend in the number of unauthenticated requests.

Security Metric over Multiple Systems (MTBUROverSystems): This security metric helps in following up the security status over enterprise IT systems as one number. Here, we apply it on the average of mean time between unauthentic requests over IT systems.

B. Approach Soundness

In this set of experiments we aim assessing our approach's accuracy and soundness in two key areas: (i) the automatic generation, deployment of security probes from input security metrics' and properties' signatures. This includes accuracy of reported measurements; and (ii) the automatic generation of the security analysis service and its results, which is based on the metrics' signatures. We figure out that we can combine the evaluation of both areas in the same experiment by testing the soundness of the whole platform –e.g. a request to a software resource R should result in a measurement M (if the security probe was generated and deployed correctly). Such measurement M should imply a change C in the value/state of a security metric or property S (if the security analysis module function as expected). Although, we did experiment on both metrics and properties, we show here evaluation results of one security property plus three different security metrics because metrics are usually more complicated in terms of calculations required – i.e. usually metrics include constraints plus aggregation functions.

A key problem we faced with these experiments is that we need to consider and enumerate different variables related to system usage and our security monitoring soundness including: *number of concurrent users*, *number of requests per second*, *number of malicious (injected faults) requests* sent to the software system, and system entities to be accessed. As a workaround solution, we conducted a set of planned experiments with different sets of variables' values to assess the security monitoring platform. In our evaluation we used a random number generator to generate random numbers for each experimental variable – i.e. number of users, user requests, malicious requests. At every time step, we generate a random number that is used to represent the number of concurrent users. Then we generate a random number for the number of requests to be issued for each concurrent user. The same applies for each experiment variable. For system entities to be accessed, we generated a hash map of some system entities with IDs. Based on the generated random number we retrieve system entity to be requested and issue a set of random valid requests and a set of malicious requests according to the total and malicious random numbers at every time step. Due to space limitations, Table 3 shows the experimental evaluation results on GalacticERP (a web-based ERP system developed internally in our research group for evaluation purposes) using one security property and three security metrics taken from Table 2. We have evaluated our approach on "Litware HR" a sample multi-tenant application built by a team at Microsoft.

TABLE 3. Security monitoring platform evaluation results

Time Step	Step 1	Step 2	Step 3	Step 4
#Users	69	56	32	84
#Requests	3429	3180	2738	4455
#Malicious Requests	2096	1921	2074	2631
Authenticated Requests **[1]**	90%	85%	58%	92%
Authentic Requests **[2]**	39%	40%	25%	41%
avg Time Between Unauthentic Requests (mSec) **[3]**	2.1	1.8	0.5	3.4
Information Disclosure **[4]**	2096	1921	1436	2631

TABLE 4. Performance Overhead of our security monitoring platform prototype implementation (values in mSec). Metrics/properties are the same from Table 3.

Metric/ property	Generate Probe	Deploy probe	Intercept Exec.	Extraction		Evaluate Metric 100 Reqs
				Time	#Records	
[1]	10	3	5	6	2	8
[2]	11	3	5	4	1	11
[3]	14	3	5	3	1	18
[4]	8	3	5	5	2	11

Table 3 shows only four time steps in our evaluation experiments. A malicious request is a request with invalid data. This depends on the property or metric being evaluated: (i) in information disclosure property, a malicious request is a request where the user is not authorized to execute the requested method. (ii) in Authenticated Requests metric, a malicious request is a request that was not authenticated. (iii) in Authentic Requests metric and Mean Time Between Unauthentic Requests metric, a malicious request is a request where the authentication control reported as an invalid request. Except for authentic requests and mean time between unauthentic requests, we did separate experiments for each metric/property at each time step. Moreover, for the authenticated request metric, we did modify the authentication control integration by MDSE@R to be applied on three methods only out of ten methods used for the evaluation experiments. The results summarized in Table 3 show that the monitoring platform successfully and accurately reported all planned violations of the information disclosure property. The same results reported for the other three operated security metrics.

C. Performance Overhead

In this experiment, we assessed the performance overhead of our security monitoring platform considering five key aspects: time to generate security probes and metric evaluation functions; time to deploy probes; overhead when intercepting system execution requests; time cost to extract system measurements required by specified security metrics; and time cost in evaluating metrics from collected security measurements. The first task is fulfilled offline without any impact on system performance. The same applies on the last one (evaluation) in case of security metrics. The deployment of probes does not impact system performance as well because we add interception points to the application configuration file at run-time. Both system interceptions for requests and measurements collections have impact on system performance. To assess the approach performance overhead, we ran a set of experiments compiling, deploying, monitoring and analyzing the set of metrics as shown in Table 2 on our Galactic multi-tenant cloud application when under heavy user loading.

Table 4 summarizes the time taken in each of these aspects in milliseconds. The time to deploy and intercept requests is relatively constant and on average takes 3 and 5 mSecs respectively. The time to generate measurement depends on the number of measurements required to verify the property or the metric (e.g. in information disclosure we need a measurement from the authorization control and a measurement from the method being accessed) and the details included in each measurement record. The analysis time depends on the complexity of the specified metric signature complexity. We measured the performance overhead of the analysis component on 100 system measurements. This is an offline task in security metrics evaluation. However, the verification of security properties requires the evaluation component to be online with the application. The performance overhead results of both metrics and properties show a very low performance overhead on application performance.

V. SUMMARY

We introduced a new tenant-oriented security monitoring approach that supports capturing and enforcing tenants' security policies and properties and assessing system security status using statistical metrics. To the best of our knowledge, this is the first approach that (i) delivers a tenant-oriented run-time security monitoring approach; (ii) combines both security properties and metrics; and (iii) uses an accessible specification language for software and security engineers. Our approach delivers a formal and familiar security metrics and properties specification language using OCL supported by a system description meta-model that helps in validating and compiling these signatures. The formalized metrics signatures are used to generate security probes that collect security measurements from a multi-tenant cloud system at run-time. They are also used in generating analysis programs that is used in analyzing the collected measurements to verify the specified security properties and assess system status. Our approach is extensible in terms of entities to be monitored and attributes to be measured at each measurement through customization of the system description meta-model. We have evaluated our approach's expressiveness compared to existing efforts, soundness in assessing and verifying such metrics, and the performance overhead of the approach. The evaluation results show that our approach is sound and expressive. It incurs very low performance overhead.

ACKNOWLEDGMENT

This research is supported by Swinburne, Swinburne Software Innovation Lab (SSIL) and NICTA (Data61) as a part of YellowBox research project.

REFERENCES

[1] M. Almorsy, J. Grundy, and I. Mueller, "An analysis of the cloud computing security problem," in *Proc. 2010 Asia Pacific Cloud Workshop, Colocated with APSEC*, Sydney, Australia, 2010.

[2] I. Ben Lahmar, H. Mukhtar, and D. Belaid, "Monitoring of Non-functional Requirements Using Dynamic Transformation of Components," in *Proc. of The 2010 Sixth International Conference on Networking and Services (ICNS)*, 2010, pp. 61-66.

[3] T. i. Holmes, E. Mulo, U. Zdun, and S. Dustdar, "Model-Aware Monitoring of SOAs for Compliance Service Engineering," in *Service Engineering*, S. Dustdar and F. Li, Eds., ed: Springer, 2011, pp. 117-136.

[4] A. J. Ramirez, B. H. C. Cheng, and P. K. McKinley, "Adaptive monitoring of software requirements," in *Requirements@Run.Time (RE@RunTime), 2010 First International Workshop on*, 2010, pp. 41-50.

[5] D. Lorenzoli and G. Spanoudakis, "EVEREST+: run-time SLA violations prediction," in Proceedings of the 5th International Workshop on Middleware for Service Oriented Computing, Bangalore, India, 2010.

[6] F. Raimondi, J. Skene, and W. Emmerich, "Efficient online monitoring of web-service SLAs," presented at the Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, Atlanta, Georgia, 2008.

[7] Roland Reichle, Mohammad Ullah Khan and Kurt Geihs, "How to combine parameter and compositional adaptation in the modelling of self-adaptive applications," presented at the PIK - Praxis der Informationsverarbeitung und Kommunikation - Special Issue: Modelling of Self-Organizing Systems, 2008.

[8] A. Amin, L. Grunske, and A. Colman, "An automated approach to forecasting QoS attributes based on linear and non-linear time series modeling," in *Proc. of the 27th IEEE/ACM International Conference on Automated Software Engineering*, Essen, Germany, 2012, pp. 130-139.

[9] F. Chen and G. Roşu, "Towards monitoring-oriented programming: A paradigm combining specification and implementation," *Electronic Notes in Theoretical Computer Science,* vol. 89, pp. 108-127, 2003.

[10] K. Havelund, G. Ro, "An Overview of the Runtime Verification Tool Java PathExplorer," *Form. Methods Syst. Des.,* vol. 24, pp. 189-215, 2004.

[11] C. Colombo, A. Francalanza, R. Mizzi, and G. Pace, "polyLarva: Runtime Verification with Configurable Resource-Aware Monitoring Boundaries," in *Software Engineering and Formal Methods*. vol. 7504, G. Eleftherakis, M. Hinchey, and M. Holcombe, Eds., ed: Springer Berlin Heidelberg, 2012, pp. 218-232.

[12] P. O. N. Meredith, D. Jin, F. Chen, and G. Roşu, "Efficient monitoring of parametric context-free patterns," *Automated Software Engineering,* vol. 17, pp. 149-180, 2010.

[13] L. Bauer, J. Ligatti, and D. Walker, "Composing security policies with polymer," *SIGPLAN Not.,* vol. 40, pp. 305-314, 2005.

[14] K. W. Hamlen and M. Jones, "Aspect-oriented in-lined reference monitors," presented at the Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security, Tucson, AZ, USA, 2008.

[15] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.,* vol. 3, pp. 30-50, 2000.

[16] S. Hussein, P. Meredith, G. Ro, "Security-policy monitoring and enforcement with JavaMOP," presented at the Proceedings of the 7th Workshop on Programming Languages and Analysis for Security, Beijing, China, 2012.

[17] Ma Jianli, Zhang Dongfang, Xu Guoai and Yang Yixian, "Model Checking Based Security Policy Verification and Validation," in *2nd International Workshop on Intelligent Systems and Applications*, Wuhan 2010, pp. 1-4.

[18] Arosha K Bandara, Emil C Lupu, and Alessandra Russo, "Using event calculus to formalise policy specification and analysis," in *IEEE 4th International Workshop on Policies for Distributed Systems and Networks.* , 2003, pp. 26-39.

[19] G. Spanoudakis, C. Kloukinas, and K. Mahbub, "The SERENITY Runtime Monitoring Framework," *Security and Dependability for Ambient Intelligence, Information Security,* vol. 45, pp. 213-238, 2009.

[20] R. M. Savola and H. Abie, "Development of security metrics for a distributed messaging system," in *Proc. of The 2009 International Conference on Application of Information and Communication Technologies*, 2009, pp. 1-6.

[21] R. M. Savola and H. Abie, "Identification of Basic Measurable Security Components for a Distributed Messaging System," presented at the Proc. of the 2009 Third International Conference on Emerging Security Information, Systems and Technologies, 2009.

[22] R. M. Savola and P. Heinonen, "Security-Measurability-Enhancing Mechanisms for a Distributed Adaptive Security Monitoring System," in *Proc. of The 2010 4th International Conference on Emerging Security Information Systems and Technologies (SECURWARE)*, 2010, pp. 25-34.

[23] R. M. Savola and P. Heinonen, "A visualization and modeling tool for security metrics and measurements management," in *Proc. of 2011 Conference Information Security South Africa (ISSA)*, 2011, pp. 1-8.

[24] A. Muñoz, J. Gonzalez, and A. Maña, "A Performance-Oriented Monitoring System for Security Properties in Cloud Computing Applications," *The Computer Journal,* vol. 55, pp. PP. 979-994, 2012.

[25] M. ı. 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.

[26] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Automated Software Architecture Security Risk Analysis Using Formalized Signatures," in *Proc. of The 36th International Conference of Software Engineering*, San Francisco, 2013, pp. 300-309.

[27] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Supporting Automated Vulnerability Analysis using Formalized Vulnerability Signatures," Swinburne University of TechnologyMarch 2012 2012.

[28] M. Almorsy, J. Grundy, and A. S. Ibrahim, "MDSE@R: Model-Driven Security Engineering at Runtime," presented at the Proc. of the 4th International Symposium on Cyberspace Safety and Security Melbourne, Australia, 2012.

[29] M. Almorsy, J. Grundy, and A. S. Ibrahim, "TOSSMA: Tenant-Oriented SaaS Applications Security Management Architecture," in *Proc. of The 5th International Conference on Cloud Computing*, Hawaii, USA, 2012, pp. 981- 988.

[30] M. Almorsy, J. Grundy, and A. Ibrahim, "VAM-aaS: Online Cloud Services Security Vulnerability Analysis and Mitigation-as-a-Service," in *Web Information Systems Engineering - WISE 2012*, X. S. Wang, I. Cruz, A. Delis, and G. Huang, Eds., ed: Springer Berlin Heidelberg, 2012, pp. 411-425.

[31] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Adaptable, Model-driven Security Engineering for SaaS Cloud-based Applications," *Automated Software Engineering Journal,* vol. to appear, 2013.

[32] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Supporting automated software re-engineering using re-aspects," presented at the Proc. of 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany, 2012.

[33] I. Avazpour and J. Grundy, "CONVErT: A framework for complex model visualisation and transformation," in *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2012, pp. 237-238.

[34] T. a. Vajk, G. Mezei, and T. e. Levendovszky, "An Incremental OCL Compiler for Modelling Environments," *In Electronic Communications of the EASST,* vol. Volume 15: OCL Concepts and Tools., 2008.

[35] OWASP. (2006). *Monitor security metrics*. Available: https://www.owasp.org/index.php/Monitor_security_metrics

[36] I. Cervesato, M. Franceschet, and A. Montanari, "A guided tour through some extensions of the Event Calculus," *Computational Intelligence,* vol. 16, pp. 307–347, 2000.

[37] B. Baliś, M. Bubak, W. Funika, R. Wismüller, M. Radecki, T. Szepieniec, *et al.*, "Performance Evaluation and Monitoring of Interactive Grid Applications," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. vol. 3241, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., ed: Springer Berlin Heidelberg, 2004, pp. 345-352.

[38] A. Vogelsang, A. Fehnker, R. Huuck, and W. Reif, "Software metrics in static program analysis," in *Proc. of the 12th international conference on Formal engineering methods and software engineering*, Shanghai, China, 2010, pp. 485-500.

[39] R. Wismüller, M. Bubak, and W. Funika, "High-Level Application Specific Performance Analysis Using the G-PM Tool," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. vol. 3666, B. Martino, D. Kranzlmüller, and J. Dongarra, Eds., ed: Springer Berlin Heidelberg, 2005, pp. 317-324.

[40] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Supporting automated vulnerability analysis using formalized vulnerability signatures," presented at the Proc. of 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany, 2012.

[41] G. Spanoudakis, C. Kloukinas, and K. Androutsopoulos, "Towards security monitoring patterns," presented at the Proceedings of the 2007 ACM symposium on Applied computing, Seoul, Korea, 2007.

[42] W. Jansen, "Directions in Security Metrics Research," NIST2009.

[43] S. Stolfo, S. M. Bellovin, and D. Evans, "Measuring Security," *Security & Privacy, IEEE,* vol. 9, pp. 60-65, 2011.

[44] M. Kamalrudin, J. Hosking, and J. Grundy, "Improving requirements quality using essential use case interaction patterns," presented at the Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu, HI, USA, 2011.