UMLtrust: Towards Developing Trust-Aware Software

Mohammad Gias Uddin
Dept. of Electrical and Computer Engineering
Queen's University, Kingston
Ontario, Canada K7L 3N6
gias@cs.queensu.ca

Mohammad Zulkernine
School of Computing
Queen's University, Kingston
Ontario, Canada K7L 3N6
mzulker@cs.queensu.ca

ABSTRACT

As users in software systems depend on each other for achieving goals, performing tasks, and utilizing resources, the trust relationships in the systems need to be considered to identify the opportunities and vulnerabilities these relationships bring. However, the problem with specifying a trust relationship is that there is no precise and a priori criteria to be satisfied. The main objective of this work is towards incorporating trust from the very beginning of a software development process. A framework is presented for specifying trust scenarios using an extension of Unified Modeling Language (UML) called UMLtrust (UML for trust scenarios). A trust scenario combines interested parties based on a context and thus helps in building a trust relationship. Suitable trust rules can be generated from the trust scenarios to monitor the trustworthiness of specific trust relationships. In this way, we can avoid conflicting, ambiguous, and redundant trust requirements in a software development life cycle (SDLC). The applicability of the approach has been illustrated using examples from file sharing applications.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design

Keywords

Trust Scenario, Unified Modeling Language (UML), Software Development Life Cycle (SDLC).

1. INTRODUCTION

Given the relationships between potential stakeholders, a system needs to be analyzed to identify the opportunities and vulnerabilities the relationships bring to the total system. Treating trust as a non functional requirement (NFR) during system development allows clarifying these relationships and reasoning about system trustworthiness. However, the problem with specifying a trust relationship is that there is no precise and a priori criteria to be satisfied. Therefore, the specification should be performed on a case-by-case basis [4]. Compared to the specifications of functional requirements (FR), there are very few tools and methodologies to support trust requirements (*e.g.*, SULTAN [1]). Moreover, most of the trust specification methodologies (*e.g.*, PolicyMaker [2], TPL [5]) only focus on authentication-based certification to build trust relationships and overlook the impact of the trust relationships from a system perspective. For example, a logged in user can still attempt to access the resource of another user illegally or can invoke a particular service in a malicious and unwanted manner. Therefore, the reasoning and specification of the trust relationships can be benefitted by incorporating different quality attributes such as competence, honesty, and security.

The Unified Modeling Language (UML) [6] is used for "specifying, visualizing, constructing, and documenting artifacts of software systems, as well as for business modeling and other non-software systems" [6]. Formal constraints to the UML models can be imposed using Object Constraint Language (OCL) [7]. A UML profile specializes UML notations for a specific domain. Although, UML provides diagrams for specifying the FR, it does not provide that much to specify the NFR including trust requirements. *"Trust (or, symmetrically, distrust) is a particular level of the subjective probability with which an agent assesses that another agent or group of agents will perform a particular action, both before he can monitor such action (or independently of his capacity ever to be able to monitor it) and in a context in which it affects his own action"* [24]. The trusting agent is called the trustor entity, and the trusted agent is called the trustee entity. The relationships between the trustor and the trustee are judged from the trustor's perspective based on a specific context. A context specifies some trust requirements that need to be satisfied to build a trust relationship. A trust scenario specifies a trust relationship including a trustor, a trustee, and a trust requirement. Based on a trust scenario, a trust rule monitors interactions between the entities. This helps in assessing the trustworthiness of the trustee according to a particular context.

In this paper, we present a UML profile called UMLtrust (UML for trust scenarios) that specializes UML notations in the domain of trust. In addition, this paper describes a trust-aware software development framework which allows developers to specify the trust scenarios using the stages of a software development life cycle (SDLC). The framework shows the gradual development of trust rules from the specified trust scenarios. The trust rules can be used for monitoring purpose and thus for dynamic decision making. The applicability of the proposed approach has been illustrated using examples from file sharing applications. The main motivation of this work is to help software developers to use UML for developing both a system and its trust scenarios. This approach of system development is useful for several reasons. First, the specification of the trust scenarios facilitates the consideration of trust early in the design process, which can be used as a driving force to guide to potential design requirements [4]. Second, the usage of the same modeling language (*i.e.*, UML) helps in avoiding any trust speci-

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

fication language only for trust purpose. Third, the development of trust scenarios can be incorporated in the traditional SDLC so that their development will not create any ambiguity and complexity. Fourth, the trust scenarios can be used to generate trust rules, which can be used to monitor the trustworthiness of the entities.

The rest of the paper is organized as follows. We start with a discussion on the state of the art of trust, system development and UML extensions in Section 2. We then present the trust-aware software development framework in Section 3. Section 4 introduces UMLtrust. Section 5 provides some examples from file sharing domain using UMLtrust. Finally, Section 6 summarizes the presented work and provides a number of future research directions.

2. RELATED WORK

Trust is incorporated into system design from three directions: specification of trust-policy to provide access to system resources [1–3, 15, 16], generation of trust-certificate to authorize someone for specific purpose [5], and inclusion of trust as a NFR in the system design by developing trust scenarios [4, 8]. A number of trust-policy languages have been proposed including SULTAN (Simple Universal Logic oriented Trust Analysis Notation) [1], PolicyMaker, KeyNote [2, 3], PICS (Platform for Internet Control Selection) [15], TPL (Trust Policy Language) [5], and REFEREE (Rule-controlled Envrironment for Evaluation of Rules and Everything Else) [16]. SULTAN focuses on trust management for internet applications by building trust relationships between entities. PolicyMaker focuses on the public key of the entity and binds it to some credentials, where a credential allows an entity to specific system environment. KeyNote is the successor of PolicyMaker with an enhancement in the verification of such policies. PICS is used to label images in the online websites, for the purpose of refraining children from watching pornography. REFEREE uses PICS to specify such access control policies. TPL considers making decisions based on the evaluation of some existing certificates about an entity. The certificates may come from trusted third parties. Although all these approaches mention access control policies for accessing certain system environment, they actually do not guide the development of a system from trust perspective. For example, they do not consider whether a system meets user requirements, or whether a system uses trust in its own development phase. Yu and Liu [4] address these issues in the requirement level of a system development by using trust as a non functional requirement, where trust is a combination of all or some quality attributes of a system. They illustrate their approach by describing the behavior of a system in the presence of attack and examine required defenses from trust perspectives. Horkoff *et al.* [8] extend their work in trusted computing base by considering that trust can be included in the technological framework of a system both in the software and in the hardware level. We support [4, 8] by capturing some system requirements such as security, competence and honesty in trust scenarios. Table 1 compares the related work with UMLtrust based on the use of trust in the system development stages.

The use of UML for the purpose of trust scenario specification has not been addressed so far. However, Górski *et al.* [9] use UML stereotypes to represent trust case. A trust case influences a trustor's level of trust by providing evidence in the form of claim, fact and assumption. The Claim Definition Language (CDL) is used to specify claims. While they use UML stereotypes and CDL to specify trust case, we use stereotypes and tagged values in use-cases and class diagrams to specify trust scenarios. Trust and risk are very much related, as risk identifies probable unwanted happening. Vraalsen *et al.* [11] use CORAS [12] language for security threat modeling. CORAS is an extended UML 2.0 profile, which uses use case diagrams to model threats and risks in the form of unwanted or malicious behavior. While CORAS works in the system requirement phase, UMLtrust focuses on the system design and specification phase. UMLQoS, a UML profile for Quality of Service (QoS) [10] specifies fault tolerant and criticality attributes. It includes constraints to specify policies. UMLintr [23] is a UML profile for specifying intrusion scenarios. While UMLintr specifies intrusion scenarios, UMLtrust specifies trust scenarios. A trust scenario may consider an intrusion scenario as part of its scenario. For example, an intrusion attempt from an entity may be considered accidental, if it happens just once. However, if the attempt happens a number of times, the entity is surely an attacker and should not be trusted. UMLsec [13] defines security requirements of a system. It employs OCL to validate any security violation. UMLtrust models trust scenarios to impose certain conditions on the system functional requirements. For example, a user can assume that a security requirement will work properly based on his conditions. These conditions are expressed in UMLtrust, where the security requirements are expressed in UMLsec. secureUML [14] models role-based access control (RBAC) by viewing a system as a set of users, roles, and resources. It is a generic security policy language from security point of view. Although they impose certain conditions on specific roles using RBAC, they do not consider the scenarios, when the conditions are violated. For example, an entity assigned to do a simple user role can attempt to gain the privilege of an administrator. Since UMLtrust encompasses these scenarios, we consider that it complements RBAC by adding certain trust requirements to the system functionalities. Table 2 compares and contrasts the other UML extensions with UMLtrust based on NFR specifications.

Table 1: Use of trust in system development

Work	Development phase	domain	Language	Trust-concern
[1]	requirement, specification	internet systems	SULTAN	trust-policy
[2, 3]	specification	application	PolicyMaker, KeyNote	trust-policy, public key
[15, 16]	specification	web-documents	PICS, REFEREE	trust-label, trust-policy
[5]	evaluation	RBAC	TPL	trust-certification
[4]	requirement	information system	*i**	trust-scenario
[8]	requirement	trusted computing	*i**	trust-scenario
Our Work	**requirement, specification**	**information system**	**UMLtrust**	**trust-scenario**

Table 2: Some UML extensions for specifying NFR

Work	Name	Specified non-functional requirement(s)
[9]	Trust case	evidence, claim
[10]	UMLQoS	fault-tolerance, criticality attributes
[11]	CORAS	security-threat modeling, risk-modeling
[23]	UMLintr	intrusion-detection
[13]	UMLsec	security policy validation, encryption
[14]	secureUML	RBAC
Our Work	**UMLtrust**	**trust-scenario**

3. THE TRUST-AWARE SOFTWARE DEVELOPMENT FRAMEWORK

Figure 1 presents the trust-aware software development framework. The right part shows the gradual development of trust scenarios, and the left part shows traditional development stages of a software system. This framework allows the software developers

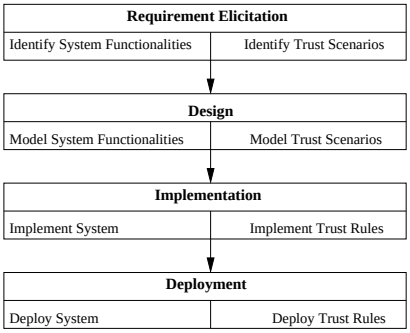

Figure 1: The trust-aware software development framework

to use UML for modeling both system behavior and related trust scenarios simultaneously. The specification of trust scenarios allows the generation of trust rules. The trust rules can be used for the monitoring purpose once the system is deployed.

To identify the trust scenarios of a system, the developers need to determine the functionalities of the system and the specific roles of different users or stakeholders. For each functionality, the specific details need to be analyzed. The analysis should include the exploration of some well-known questions. Some example questions are as follows [4]. First, what is expected from a specific functionality? How it can be achieved? Second, what is the outcome of using the functionality? Third, what can go against the functionality? Fourth, what is the outcome in terms of negative consequences? This analysis identifies the possible trust requirements from the functionalities. The use case diagrams are used to represent different trust scenarios. The class diagrams model the static attributes of the scenarios. Moreover, the possible system states of the trust scenarios are identified using the state-machine diagrams. The class diagrams and the state-machine diagrams can be used to implement the trust scenarios. The implementation of the trust scenarios provide the trust rules.

4. UMLtrust: THE UML PROFILE

UML can be extended to particular domain in two ways. First, UML can be extended in the meta meta level by creating, deleting or modifying current UML syntax and semantics. An example of this extension is Rational Rose Real Time [17]. Second, suitable stereotypes and tagged values can be added by extending UML meta models. The stereotypes specialize the UML diagrams to specific domain. The tagged values add specific information to the stereotypes. UMLtrust adopts the second approach.

We use use-case, class, and state-machines diagrams. The use-cases represent trust scenarios by focusing on the customer or the managerial point of view. The class diagrams specify the static attributes and the structure of the trust scenarios. The state-machine diagrams present the dynamic nature of the trust scenarios. In addition, the package diagrams store similar types of trust scenarios. We name our use-cases as trust-use-cases, class diagrams as trust-class diagrams, state-machines as trust-state-machine diagrams, and package diagrams as trust-package diagrams.

Stereotypes and tagged values. The stereotypes and tagged values of UMLtrust are provided in Table 3 and Table 4 respectively. The first six stereotypes are for trust-use-case diagrams representing trustor, trustee, and different ways of having trust relationships between them. The `Connector` type stereotypes are used to imply the impact of different connections in the trust-use-case diagrams. The next three stereotypes are used in the trust-class diagrams.

Table 3: UMLtrust stereotypes

Stereotype	Base Class	Description
$<< trustor >>$	Actor	trusting actor
$<< trustee >>$	Actor	trusted actor
$<< trust-service >>$	Use case	provided service
$<< trust-resource >>$	N/A	shared resource
$<< trust-cert >>$	N/A	trusted property in certificate
$<< trust-infr >>$	N/A	trusted base infrastructure
$<< trusts >>$	Connector	trusting behavior
$<< owns >>$	Connector	certificate ownership
$<< holds >>$	Connector	holding/controlling a resource
$<< provides >>$	Connector	providing a service
$<< uses >>$	Connector	actions performed by trustee
$<< exploit >>$	Connector	
$<< trustor >>$	Class	trustor class
$<< trustee >>$	Class	trustee class
$<< trust-concern >>$	Class	concern related to a trust-context of a trust relationship
$<< trust\ rules >>$	Package	package to store trust-rules

Table 4: UMLtrust tagged values

Tagged Value	Class	Description
goal	$<< trustor >>$	The goal to be achieved from a trust relationship
min-trust-level	$<< trustor >>$	Minimum level of trust required to form a trust relationship (*e.g.*, low, medium. high)
req-attr	$<< trustor >>$	The required quality expected from a trust relationship (*e.g.*, competence, security, honesty)
level	$<< trust-concern >>$	The direction of a trust relationship from $<< trustor >>$ to $<< trustee >>$ (*e.g.*, u-to-u for user to user, u-to-s for user to system, s-to-u for system to user)
category	$<< trust-concern >>$	The specific category (*i.e.*, context) of a trust concern.
life-time	$<< trustee >>$	Maximum life time of a trust value calculated about a relationship (*e.g.*, long, short)
method	$<< trustee >>$	Methods used in a trust relationship (*e.g.*, provides, exploit)

Trust-use-cases. A trust-use-case consists of a $<< trustor >>$ actor and a $<< trustee >>$ actor. The connection between them provides a trust relationship. The trust relationships are categorized into four forms [1]:

- $<< trust-service >>$: A $<< trustor >>$ trusts the quality of a service provided by the $<< trustee >>$. On the other hand, a $<< trustor >>$ trusts a $<< trustee >>$ to invoke services provided by the $<< trustor >>$ in a trustworthy manner (*e.g.*, without exploiting the service).

- $<< trust-resource >>$: A $<< trustor >>$ trusts a $<< trustee >>$ to use resources that it owns or controls. The resource provider can be the system itself or individual users in the system. For example, a software execution environment or an application service can be $<< trust-resource >>$.

- $<< trust-cert >>$: A $<< trustor >>$ may trust a $<< trustee >>$ based on a certification provided by a third party. In a software system, this certification can be treated as the reputation of the $<< trustee >>$ or the valid user-id.

- $<< trust-infr >>$: A $<< trustor >>$ has to trust some base infrastructure which the system offers (*e.g.*, the hardware used by $<< trustor >>$ or the operating system on which the system is running).

Figure 2 provides the iconic representations of $<< trust-cert >>$,

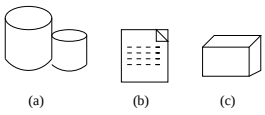

Figure 2: Representative icons for stereotypes (a) $<< trust - resource >>$**, (b)** $<< trust - cert >>$ **(c)** $<< trust - infr >>$

$<< trust - resource >>$, and $<< trust - infr >>$. The iconic representations are used to facilitate them in the trust-use-cases.

The `Connector` type stereotypes designate different trust relationships in the trust-use-case diagrams. $<< trusts >>$ denotes trusting behavior of a $<< trustor >>$. $<< owns >>$ implies the ownership of particular certification. $<< holds >>$ represents holding or controlling of a resource. The provision of a service is represented by $<< provides >>$. $<< uses >>$ specifies the usage of a specific methodology to achieve something (*e.g.*, achieving a certification). $<< exploit >>$ designates the risk behavior of a trust-use-case, which the $<< trustee >>$ uses to deceive or harm the $<< trustor >>$. The exploitation can have many forms, and it differs depending on the scenarios.

Trust-packages. There is only one stereotyped package in UMLtrust, $<< trust - rules >>$. Different systems have different types of trust rules. Therefore, the $<< trust - rules >>$ package should be categorized based on a system. For example, for a system with M categories of trust rules, there will be M types of $<< trust - rules >>$ packages, each with different names. This helps storing the trust rules in the packages based on their similarity.

Trust-classes. Three types of stereotyped classes are used, $<< trustor >>$, $<< trustee >>$, and $<< trust - concern >>$. $<< trustor >>$ represents the trusting entity, which uses some tagged values to imply some trust related information. These information are necessary to build the trust relationships. The tagged values for $<< trustor >>$ class are `goal`, `min-trust-level`, and `req-attr`. `goal` is used to identify the specific goal the $<< trustor >>$ wants to achieve from a trust relationship. `min-trust-level` provides the minimum trust level that is needed from a $<< trustee >>$ to build a trust relationship. `req-attr` implies the trust attribute that should be present to build the trust relationship (*e.g.*, competent to perform a task, secure to protect something, or honest to some perspective). $<< trustee >>$ class represents the trusted entity, which has one tagged value, `method`. This tagged value is used to identify the intention of the $<< trustee >>$ while building a trust relationship. The $<< trust - concern >>$ class encapsulates the specific concern based on which a trust relationship should be formed. This class uses specific attributes which are needed to be monitored. This class has three tagged values: `level` is used to identify the type of the trust relationship that is being addressed by this class (*e.g.*, `u-to-u` is for trust relationship between two users, `u-to-s` and `s-to-u` are for trust relationship between the users and the system itself, where `u-to-s` tells that the trustor is a user and the trustee is a system). `category` is used to categorize the $<< trust - concern >>$ class to a specific context. This is necessary to identify the package of the corresponding trust rules. `life-time` designates the default activation time of the corresponding trust rule based on the $<< trust - concern >>$. This tagged value is necessary for refreshing the trust values of a trust relationship that is necessary for dynamic decision making.

Trust-state-machines. We use the standard state-machine diagrams to represent the trust-scenarios. However, we include three types of final states in our state-machine diagram: successful ($I(S)$), unsuccessful ($I(U)$), and alarming ($I(A)$). A successful final state of an interaction is reached, when no trust-violation is occurred. In an unsuccessful final state, a violation of trust is detected. An alarming state is identified, when the $<< trustee >>$ has done something suspicious but not completely untrustworthy.

Table 5: Elicited trust concerns for file sharing server [19–21]

Trust scenarios	Description
Illegal access attempt	The clients may try to get access to another client's personal files in the resource database
File spamming	The clients may upload illegal and insignificant files to waste the storage space of the server.
Excessive file size	The clients may try to upload files beyond the limit of the server to make the upload service unavailable for others
Write harmful file	The clients may upload files containing virus or malicious scripts
Remote file inclusion	The clients may manipulate the file open service of the file sharing server to open malicious files remotely and to execute it on the server

5. EXAMPLE TRUST SCENARIOS

We adapt file sharing into a conceptual file-storage grid system [18] to illustrate the applicability of UMLtrust. In a file storage grid, there are multiple distributed servers. Since the servers are publicly accessible, the users (clients) use the servers mainly for uploading and sharing resources (*i.e.*, files) with each other. It is obvious that not all the users are benevolent, and so they may exploit the file-sharing services for their own purpose. Therefore, trust concerns should be incorporated in the development of such a file sharing system to monitor the trustworthiness of the clients. Based on the trust monitoring, the server can decide which client entity should be granted services or resources. A file sharing server offers three basic services to its clients, file uploading, sharing, and downloading. We analyze the services from trust perspective, and elicit some trust requirements using the documentations available for file sharing applications [19–21]. Some elicited trust concerns are provided in Table 5. We present the modeling of the first two scenarios (*i.e.*, illegal access attempt and file spamming) using UMLtrust in the next two subsections. However, the rest of the trust scenarios can also be modeled using UMLtrust.

5.1 Trust Scenario 1: Illegal Access Attempt

An advantage of sharing resources through public resource grid is that the grids are distributed, and the failure of one grid will not close down the whole file storage system. However, since the grids are publicly accessible, there is always the risk of clients maliciously trying to access the resource of other clients. A popular and well known attempt to do this task is the SQL injection attack [22] against the search engine of the grid storage system. The attack is performed by maliciously crafting search query, which may grant a normal client the access to other clients resources. A client is untrustworthy, if it attempts to access such resources. We encapsulate such illegal access attempt behavior in this trust scenario. This scenario needs to be updated, if any illegal access attempt other than the SQL injection attack happens against the system.

Trust-use-case. The trust-use-case for the illegal access scenario is presented in Figure 3. The $<< trustor >>$ is `anyFileStorage` server. The $<< trustee >>$ is `anyFileUser`. The $<< trustor >>$ trusts the $<< trustee >>$ in using the database. The $<< trustee >>$ exploits the `PersonalFileDatabase` of the file server in order to gain unauthorized access to the resources of the other parties.

Trust-class. Figure 4 provides the class diagrams to specify the illegal access attempt from trust perspective. The `goal` of the $<< trustor >>$ is `confidentiality` and `integrity`, as the server needs to protect the resources from unauthorized parties. The `req-attr` is

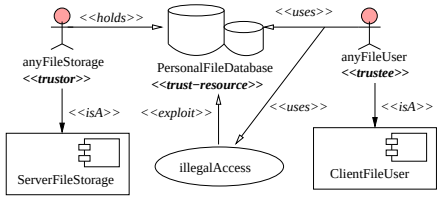

Figure 3: Trust-use-case diagram for illegal access attempt

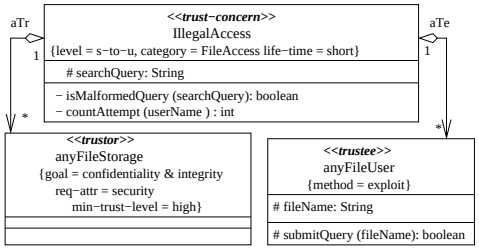

Figure 4: Trust-class diagram for illegal access attempt

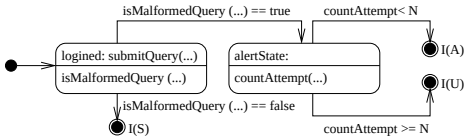

Figure 5: Trust-state diagram for illegal access attempt

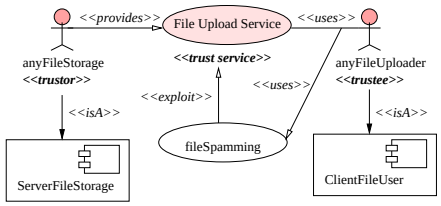

Figure 6: Trust-use-case diagram for file spamming

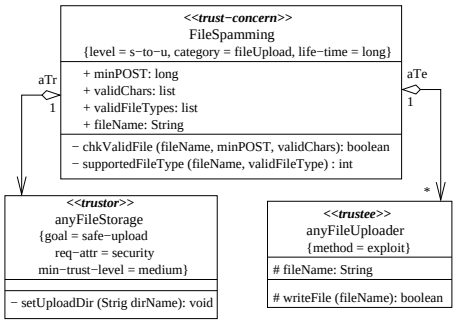

Figure 7: Trust-class diagram for file spamming

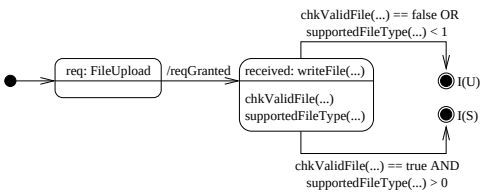

Figure 8: Trust-state-machine diagram for file spamming

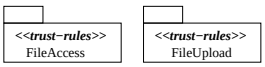

Figure 9: Trust-package diagrams for illegal access attempt and file spamming

`security`. The `min-trust-level` is `high`. The $<< trustee >>$ exploits (`method`) the search query while submitting the query using `submitQuery(...)`. The $<< trust - concern >>$ is `IllegalAccess` with `level` set to `s-to-u` and `category` set to `FileAccess`. The `life-time` is `short`, which implies the effective activation time of the corresponding trust rule. `isMalformedQuery(...)` takes the `searchQuery` as parameter and checks any attempt of the SQL injection attack in the query. Since users may accidentally input such malformed query for once, the function `countAttempt(...)` is used to keep track of the number of times a user has tried to input maliciously crafted queries. If this number of attempts goes beyond an acceptable limit, the user is regarded as untrustworthy.

Trust-state-machine. The trust-state-machine diagram of the illegal access attempt is provided in Figure 5. The user in the login state searches for files in the file database. If the user inputs normal query, the interaction with the user is considered satisfactory ($I(S)$). However, if the user inputs malformed search query, the alert state is triggered. Then each such attempt is counted. For attempts beyond the acceptable limit (*e.g.*, N), the user is considered untrustworthy and the corresponding interaction is treated as unsuccessful ($I(U)$). For attempts less than N times but more than once, the interaction remains in the alarming state ($I(A)$).

5.2 Trust Scenario 2: File Spamming

The untrustworthy clients may have malicious intentions, while using the file uploading service of the file-server. They can attempt to spam the server by uploading illegal and invalid files for the purpose of (i) wasting storage space of a server and (ii) making careless use of the file uploading service. The client can run spam script to upload such kinds of files [21]. The server can control this activity by putting certain constraints in the file uploading service.

Trust-use-case. Figure 6 presents the trust-use-case diagram representing the file spamming scenario. `anyFileUploader` is the client, and `anyFileStorage` is the server. The server trusts a client by providing it the service `fileUpload`, and the client invokes ($<< uses >>$) it. However, the client can upload malicious and unwanted contents ($<< exploit >>$) by using `fileSpamming`.

Trust-class. The class diagrams for the file spaaming scenario are provided in Figure 7, where the `goal` of the $<< trustor >>$ is to ensure `safe-uploading`. It requires that the interaction with the $<< trustee >>$ is secure with `min-trust-level` set to `medium`, as spamming of insignificant files may not be the greatest risk to the server. The $<< trustee >>$ attempts to violate the trust of the $<< trustor >>$ by exploiting the `writeFile(...)` command. The $<< trustor >>$ opens a local directory (`setUploadDir(...)`), where the client uploads files. `writeFile(...)` is used to upload the files to the server. The $<< trustee >>$ can succeed in file spamming, when the $<< trustor >>$ does not check the uploaded files. The file spamming happens when the $<< trustee >>$ writes files that are very small in size (`minPOST`, *i.e.*, smaller than any acceptable size for a particular file type), contain unsupported characters in the filenames (*e.g.*, &, !, ^, ; based on `validChars`), or unsupported by the file server (*e.g.*, for a document file server, a movie file is a spam according to `validFileTypes`). These attributes are defined in $<< trust-concern >>$ class, where the `level` is `s-to-u` for system to user trust, and `category` is `FileUpload`. If we consider service

as a particular type of context for building a trust relationship, then `FileUpload` is the service name, and `writeCommand(...)` is the corresponding system functionality. The `chkValidFile(...)` function checks the file names. The `supportedFileType(...)` function checks whether the file type meets the server requirements.

Trust-state-machine. Figure 8 captures the states necessary to represent the file spamming scenario. In the first state, the client requests for uploading. Upon the granting of the request, the client uploads file (`writeFile(...)`). Once the file is uploaded, the functions `chkValidFile(...)` and `supportedFileType(...)` are called. The first function returns true, if the file is valid, and the second function returns a value greater than 0, if the file is supported by the server. Trust is violated, if the file is found as spam (*i.e.*, not valid or not supported). Then the final state is reached as unsuccessful ($I(U)$, *i.e.*, the interaction between the server and client was unsuccessful). The interaction is denoted as successful, $I(S)$, when both the functions identify the uploaded file as valid.

Figure 9 presents the trust-packages `FileAccess` and `FileUpload` to store the illegal access attempt and the file spamming trust scenarios respectively.

6. CONCLUSIONS

In this work, we model trust-scenarios using UMLtrust (UML for trust scenarios), a UML extension specialized for the domain of trust. Suitable trust rules can be developed from the trust scenarios to monitor and calculate trust once the system is deployed. A framework is proposed to incorporate the development of trust scenarios and trust rules in the traditional SDLC. This helps in locating and modeling trust scenarios along with the system development. The incorporation of trust scenarios to the total software system development can make the system trust-aware, robust, and expectation specific. Examples using the file sharing applications are provided to illustrate the applicability of UMLtrust.

The trust scenarios considered in this paper, however, focus on expectations and beliefs in the trust-establishment process and overlook authentication requirements. Nevertheless, binding the two concepts together can be more effective. Future work in this direction is to model more systems of variable natures to utilize the effectiveness of UMLtrust. The calculation of trust requires a specific mechanism, which we leave open in this work.

Acknowledgement

This research is partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

7. REFERENCES

[1] Grandison, T. Trust management for internet applications, *PhD Thesis*, University of London, July 2002.

[2] Blaze, M., Feigenbaum, J., Lacy, J. Decentralized trust management, in *Proc of the 1996 IEEE Symposium on Security and Privacy*, Oakland, USA, 1996. IEEE CS Press: 164–173.

[3] Blaze, M., Feigenbaum, J., Keromytis, A. KeyNote: Trust management for public-Key infrastructures, in *Proc of the 6th Int Workshop on Security Protocols (LNCS v1550)*, UK, 1998. Springer: 625–629.

[4] Yu, E., Liu, L. Modelling trust for system design using the *i** strategic actors framework in *Proc of the Int Workshop on Deception, Fraud, and Trust in Agent Societies (LNCS v2246)*, Barcelona, Spain, 2001. Springer-Verlag: 175–194.

[5] Herzberg, A., Mass, Y., Michaeli, J., Naor, D., Ravid, Y. Access control meets public key infrastructure, or: assigning roles to strangers, in *Proc of the 2000 IEEE Symposium on Security and Privacy*, Oakland, USA, 2000. IEEE CS Press: 2–14.

[6] OMG. OMG Unified Modeling Language Specification, Technical report, Object Management group, March 2003.

[7] OMG. OMG Object Constraint Language Version 2.0, Technical report, Object Management group, May 2006.

[8] Horkoff, J., Yu, E., Liu, L. Analyzing Trust in Technology Strategies, in *Proc of the Int Conference on Privacy, Security, and Trust*, Ontario, Canada, 2006. McGraw-Hill: 21–32.

[9] Górski, J., Jarzebowicz, A., Miler, J., Olszewski, M. Trust case: justifying trust in an IT solution, in *Reliability Engineering & System Safety: Elsevier*, 2004. **89**(1): 33–47.

[10] OMG Management Group. UML^{TM} profile for modeling quality of service and fault tolerance characteristics and mechanisms, Technical Report, 2004.

[11] Vraalsen, F., Lund, M., Mahler, T., Stlen, K. Specifying legal risk scenarios using the CORAS threat modelling language, in *Proc of the 3rd Int Conference on Trust Management (LNCS v3477)*, Rocquencourt, 2005. Springer: 45–60.

[12] Raptis, D., Dimitrakos, T., Gran, B., Stølen, K. The coras approach for model-based risk management applied to e-commerce domain, in *Proc of the IFIP TC6/TC11 6th Joint Working Conference on Communications and Multimedia Security*, Slovenia, 2002. Kluwer, B.V.: 169–181.

[13] Jürgens, J. Secure system development with UML, Springer-Verlag, 2003.

[14] Basin, D., Doser, J., Lodderstedt, T. Model-driven security for process-oriented systems, in *Proc of the ACM Symposium on Access Control Models and Technologies*, Como, Italy, 2003. ACM Press: 100–109.

[15] Resnick, P., Miller, J. PICS: Internet access controls without censorship, in *Communications of the ACM*, 1996. **39**(10): 87–93.

[16] Chu, Y., Feigenbaum, J., LaMacchia, B., Resnick, P., Strauss, M. REFEREE: Trust Management for Web Applications, in *World Wide Web Journal*, Summer 1997; **2**(3): 127–139.

[17] Selic, B., Rumbaugh, J. Using UML for modeling complex real-time systems, Rational (ObjectTime), March 1998.

[18] Dimmock, N., Belokosztolszki, A., Eyers, D., Bacon, J., Ingram, D., Moody, K. Using trust and risk in role-based access control policies in *Proc of the 9th ACM symposium on Access control models and technologies*, New York, USA, 2004. ACM Press: 156–162.

[19] EUROSEC GmbH Chiffriertechnik & Sicherheit, Secure programming in PHP, in *Secologic Project*, 2005.

[20] Bezroutchko, A. Secure file upload in PHP web applications, in `http://www.scanit.be/uploads/php-file-upload.pdf`, 2007. (August 2007)

[21] File upload. in `http://ikiwiki.info/todo/` (August 2007)

[22] Snyder, C., Southwell, M. Preventing SQL injection, in *Pro PHP Security*, 2005. SpringerLink: 249–261.

[23] Hussein, M., Zulkernine, M. UMLintr: a UML profile for specifying intrusions in *Proc of the 13th IEEE Int Conference and Workshop on the Engineering of Computer Based Systems*, Potsdam, 2006. IEEE CS Press: 279–286.

[24] Gambetta, D. Can we trust trust?, in *Trust: Making and Breaking Cooperative Relations*, Gambetta, D. (ed.), Chapter 13, 1988. University of Oxford: 213–237.