CERIAS Tech Report 2005-67

SOFTWARE ENGINEERING FOR SECURE SOFTWARE - STATE OF THE ART: A SURVEY

by Jayaram K R and Aditya P Mathur

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Software Engineering for Secure Software - State of the Art: A Survey

Jayaram K R and Aditya P. Mathur

Department of Computer Science

Purdue University

West Lafayette, IN 47907, USA

E-mail: {jayaram,apm}@purdue.edu

September 19, 2005

Abstract

This report contains a survey of the state of the art in software engineering for secure software. Secure software is defined and techniques used in each phase of the software lifecycle to engineer the development of secure software are described. Also identified are open questions and areas where further research is needed.

The survey reported here was undertaken to understand how the practice of software engineering blends with the requirement of secure software. This has resulted in a novel two-dimensional description of the relationship between the software lifecycle phases and techniques for satisfying security requirements. The report is organized around this relationship.

Contents

List of Figures

List of Tables

1 Introduction

Computer Security is fast becoming an important issue in many areas. Traditionally, security has always been treated as an add on. An application was assumed to be secure if it used cryptography, security protocols, etc. Security Mechanisms were treated as silver bullets. Attacks on protocols made the community realise that *intuitive* correctness was insuffi cient for security. This resulted in the development of Formal Verifi cation Techniques for Security Protocols. But, Formal Verifi cation has its own disadvantages. First, it could only verify protocol design, it has theoretical limitations and could not guarantee the security properties of protocol implementations. With the proliferation of hackers, who exploit *software defects*, especially mundane defects, to compromise systems, security aspects of software has become an area of concern. Security is now a feature of the system as a whole. Researchers are concerned about security software–software whose primary functionality is to implement a security protocol or mechanism, and security of software–software that functions correctly under malicious use and that which does not contain loopholes.

Verifi cation and Static Analysis, which deals with the analysis of programs for common security flaws, after they are built, are not effective in ensuring security though we acknowledge that a lot of progress has been made in analyzing artifacts. The complex ways in which security permeates systems has led researchers to believe that software engineering will be much more effective in ensuring security. This is because Software Engineering for Secure Software focusses on the top-down approach to building secure software. From a software engineering perspective, one is interested in how the (1) existing lifecycle phases, (2) artifacts and (3) techniques used in each phase should be augmented (or perhaps new techniques introduced?) to support security. The holy grail of this fi eld is software which is *secure* by construction. In the words of Thomas Ball [Bal05], "A program is a very detailed solution to a much more abstract problem. Leading from a problem to a program is a complex process." We believe that security will improve only by focussing on this process.

Security is a broad area. It deals with cryptography, security protocols, access control, information flow, software security, program obfuscation, etc. In Section 2, we provide a classifi cation of security properties to simplify our presentation. Research efforts in this area can be organized into a matrix as in Table 1, with rows representing phases of the software lifecycle and columns representing various security concepts. We do not have a complete recommended lifecycle to engineer secure software. Research focused on specifi c lifecycle phases is surveyed here. Empty cells in the table indicate areas where we were unable to fi nd any references. We restrict the scope of this report to the security concepts outlined in Table 1. In particular, we do not cover privacy and security certifi cation. The citations in this report are intended to be representative of the state-of-the-art and not thorough in any sense.

Table 1: Software Engineering for Secure Software

Research	Requirements Engg. and Specification	Analysis and Design	Implementation	Testing
Software Security	3, 6.4 6.1	3 7.2	3 8	3 9.1
Security Protocols	6.2 5	7.1		9.2
Firewalls				9.3
Intrusion Detection				9.4
Obfuscation				
Access Control	6.3	6.3.3	6.3.3	

2 Background

The term security may either mean Software Protection, Software Security or Information Security.

Software Security: According to McGraw [CM04a], software Security is engineering software to function correctly under malicious attack. Secure Software is software that does ONLY what it is supposed to do and nothing else. Reliable software, on the other hand does what it is supposed to do but may also exhibit some un-specifi ed behavior. Un-specifi ed behavior is often harmful. Memory leaks, buffer overfbws and other common attack patterns used by hackers are the results of un-specifi ed software behavior. *Engineering Secure Software* involves taking a pro-active approach. It is not an add-on collection of techniques. It is different from security software. It is a feature of the entire software system and cannot be ensured by using security mechanisms like access control, encryption, SSL, etc.

Software Vulnerabilities and Risk: Software Security is affected by software vulnerabilities. A vulnerability is either a defect, or a bug, or a flaw. Defects are implementation and/or design errors. A defect may lie dormant in software for several years and then surface in a fi elded system with major consequences. A bug is an implementation-level software error. Normally, *bugs* refer to low-level implementation errors that can be remedied by limited code analysis of the external environment. A flaw is s subtle defect at a deeper level. Risk is the probability that a defect or a bug or a flaw is actually manifested resulting in either an impact on normal software functioning or a failure. Risk is normally

coupled with a particular vulnerability though sometimes, a set of vulnerabilities may result in a heightened risk.

Software Protection: Software Protection involves (1) protecting software from tampering by a malicious host, (2) security of code as an artifact and (3) security of run-time program behavior. By security of run-time program behavior, we mean securing the purpose, control, and data-fbw of parts of code. Sometimes, it is essential to prevent reverse engineering of code. This is usually done by obfuscation [CTL97]. Thus, Code Obfuscation refers to the process obscuring programs. Software implementing Obfuscation techniques is also called obscurity software. Obfuscation being a security mechanism; obscurity software is also security software.

Code Obfuscation is the process by which source code is transformed into an equivalent one, i.e. the transformed code retains the same functionality as the original but is substantially more diffi cult to understand and reverse engineer. Obfuscation is done either by:-

1. Lexical Transformations - that modify only the lexical structure of the program, for example - scrambling identifi ers. Such transformations are generally not effective in that code so transformed is relatively easy to reverse engineer.
2. Control Transformations - that modify the control fbw of the program [CTL98b].
3. Data Transformations - that obfuscate data structures used [CTL98a]

Another approach to software protection is tamper-proof code, i.e. code, in which additional logic is inserted after compilation, to detect tampering and respond to it. See for example, the work by Atallah and Chang [CA01].

Information Security: Information Security refers to the *Security of Information* i.e. confi dentiality, integrity and control over information and/or resources. Information Security includes Network Security, Access Control and Secure Information Flow. *Network Security* refers to mechanisms used to secure information on the network, typically cryptographic protocols which include mechanisms like PGP, SecureEmail, SSL, SSH, SCP, SecureFTP, etc. or a combination of these. Network Security also includes mechanisms to avoid software exploits such as fi rewalls,reactive mechanisms such as intrusion-detection systems. Denial of Service and its prevention also falls under Network Security. Access Control includes delegation, authorization, trust management etc. We place Trust management under Network Security as well e.g. trust establishment, trust negotiation etc.

It is useful to examine how these concepts relate to each other. An attacker may take advantage of a software vulnerability to compromise confi dential information. Information security mechanisms, such as protocols and access controls, are eventually implemented

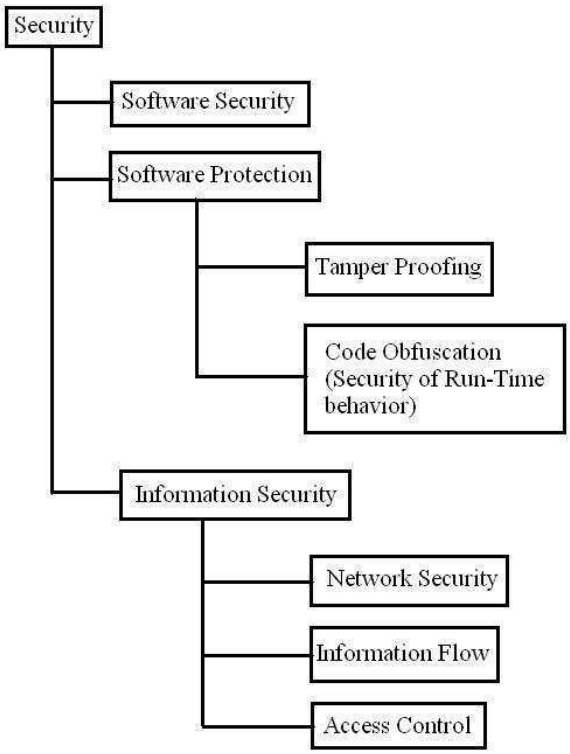

Figure 1: Security Classification

resulting in security software. Security Software may have vulnerabilities and we thereby get into Software Security of Security Software or *Secure Security Software*. It is now apparent that the word *security* is overloaded and has a different meaning in each context. We wish to add that the classifi cation in Figure 1 is not a unique or universally accepted. We use it as terminology in this report.

3 Software Security

Research has focussed on using a risk management approach to software security. Good software security practice leverages good software engineering practices: thinking about security early in the software lifecycle, knowing and understanding common threats (including language-based flaws and pitfalls), designing for security, and subjecting all software artifacts to thorough objective risk analyses and testing. Complicating the software security problem are reliance on networked systems and devices, easy extensibility and increased complexity of systems.

Security Development Lifecylce (SDL) Research in Software Security Engineering has focussed on using so-called *best practices* in the software lifecycle. As the name implies, a security development lifecycle is a software development lifecycle where a special emphasis is placed on software security in each phase. Two SDLs have been proposed to integrate software security into the lifecycle, one is by Microsoft as part of its Trustworthy Computing Initiative [HL03] and the other by McGraw [CM04a]. We refer to these efforts as Microsoft's SDL and McGraw's SDL, respectively. Figures 2 and 3 provide a pictorial overview of the two SDLs. As is evident, both SDLs have a lot in common.

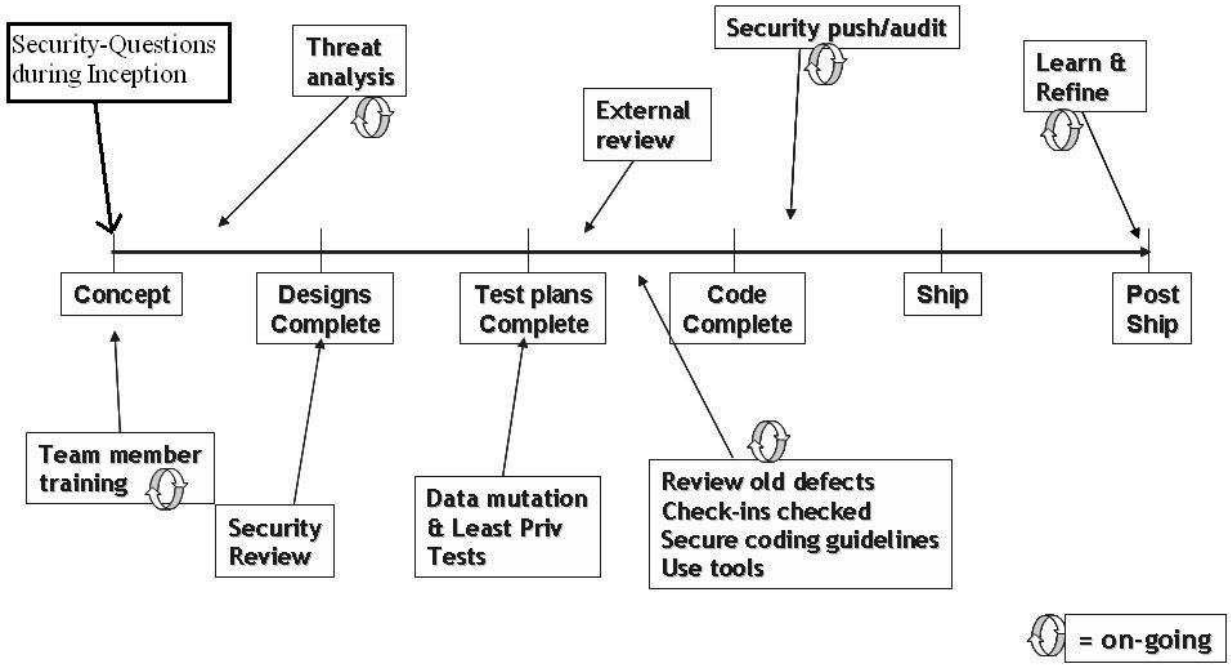

Figure 2: Microsoft's SDL

4 Model-based Security Engineering

Model-based security engineering, or Model-driven Security, is an approach to building secure systems in which software engineers:-

- Formally specify the system requirements and other artifacts, e.g. design.
- Analyze models automatically against their security requirements, often using formal analyses such as theorem proving, model checking and static analysis.
- Generate as much code as possible from the model.
- Generate tests from the model to test the fi nal implementation

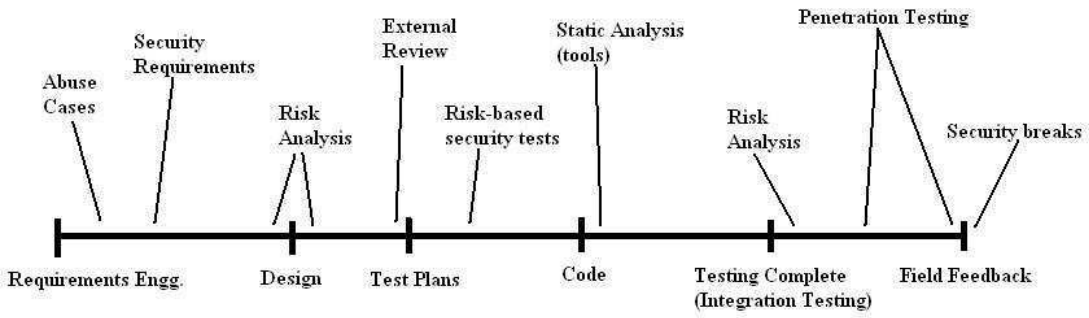

Figure 3: McGraw's SDL

- Generate models from the source code for understanding and analyzing legacy systems i.e. $code \Rightarrow model$.

It has become a common practice in software engineering to use models. UML models are used in many development processes. Models are precise specifi cations of requirements and design. They enable analysis, both formal and informal, and signifi cantly increase chances of early fault detection. They serve as specifi cations for successive software engineering phases and, when suffi ciently formal, they can be used to generate code.

Models are used in security and policy specifi cations. However, security models and system design models are typically disjoint and are expressed in different ways. In general, integration of system design models with security models is poorly understood and consequently, not adequately supported by most modern software processes and tools. While the fully automated generation of a complex system from its model appears infeasible, it is possible to automate the generation of platform-specifi c support for certain system concerns, in particular access control, persistence and logging.

The central artifacts that support Model-driven security are security design models which combine security requirements and design. Rather than present one particular modeling language for constructing these models, Lodderstedt and Basin [BDL04] propose a schema for building such languages in a modular way. A language for constructing security design models should have:-

1. A security modeling language for expressing security requirements and policies.
2. A systems design modeling language for constructing system models.
3. A dialect which combines (1) and (2) so that expressions in (1) can make statements about model objects in (2)

5 UMLSec

UMLSec [JÖ2] is an extension of UML for Secure Systems Development. Research by Jan Jürjens [JÖ3] is a partial realization of the Model-driven security framework. UMLSec expresses security-relevant information within UML System Specifi cation diagrams such as interaction and deployment diagrams.

UMLSec is defi ned as a UML profi le using three lightweight UML extension mechanisms namely stereotypes, tagged values and constraints. A stereotype defi nes a new type of modeling element by extending the semantics of the UML metamodel. Some UMLSec stereotypes are *link, Encrypted link, critical object, and fair exchange*. Note that every UML profi le has a fi nite and fi xed set of stereotypes associated with it. A stereotype has variables. A tag is an instantiation of a variable. Each stereotype has tags, threat specifi cations and constraints associated with it.

Constraints specify security requirements and threat specifi cations model actions taken by the adversary. Different threat scenarios can be specifi ed based on adversary strengths. UMLSec also provides cryptographic primitives such as encryption, decryption, and message digest. An important point to note is that UMLSec has stereotypes and constructs to model the underlying cryptographic system. Hence, tags and constraints that specify security requirements can directly make statements about system components. Thus, in UMLSec, knowledge of prudent security engineering has been encapsulated into stereotypes so that they can be used in system modeling by developers.

6 Requirements Engineering and Specification

6.1 Abuse Cases

Abuse Cases, also known as Anti-Requirements or Misuse Cases, were introduced by McDermott and Fox [MF99] and have been extensively studied by Crool et al. [CILN02], Alexander [Ale04, Ale02, Ale03] and Hope et al. [HMA04].

A use case depicts the functionality of a system. It describes a complete transaction between one or more actors and the system. Abuse cases represent ways in which a system can be improperly used. It is important to note that an abuse case *must* be described in terms of (1) transactions resulting in actual harm and (2) abuse of privilege associated with each transaction.

One can never be certain as to where a flaw might occur or whether the attacker employs minimal abuse of privilege necessary. Hence, an abuse case represents a family of transactions, with each family member associated with a range of privileges needed for the abuse. They can be represented by UML use case diagrams and are usually described in a natural language. See [MF99] for examples. Hope et al.[HMA04] advocate using attack patterns to construct abuse cases. Akin to design patterns, attack patterns describe tech-

niques in an abstract way commonly used by hackers to attack software. An exhaustive documentation of attack patterns is available in a book by Hoglund and McGraw [HM04].

Abuse cases make a System Engineer explicitly consider ways in which every new requirement or feature can be attacked, at its conception. It is important to remember that an abuse case is a generic requirements engineering concept and is not specifi c to any security property or to software security. However, abuse cases cannot exhaustively describe ways in which a software system can be attacked because they are manually derived based on existing knowledge of attack patterns, hacks, etc.

6.2 Requirements Specification for Security Protocols

There has been a signifi cant amount of research in requirements specifi cation in the context of *Formal Methods for Security protocols.* Meadows [Mea03a] provides a survey and Syverson and Meadows [Mea03b, SM96] consider trends in requirements specifi cation. The term *Formal Methods* normally refers to a combination of:-

1. Use of a mathematical or a logical model, called formal model, of the security protocol along with its security requirements specifi ed mathematically, and,
2. an effective and tractable procedure to determine whether the protocol satisfi es its requirements

One approach in the formal analysis of security protocols is to analyze all feasible traces of the state transition system (STS) and to determine whether or not the security properties are preserved for each trace. Many Network security related properties are trace properties such as secrecy, authentication, and fair exchange. If the protocol running in parallel with the attacker is viewed as an STS, the protocol analysis problem for trace properties can be stated as a reachability problem, i.e. the problem of determining if the state in which the security property is violated is reachable from the protocol's initial state. This is usually done by model checking though other state-space exploration techniques also exist. In this survey, we focus on formal specifi cations and their expressivity. The reader is referred to [Mea03a] for a recent survey of verifi cation methods.

6.2.1 Specifying Secrecy

Usually, each state of the STS has a set K which stores the current knowledge of the adversary. Any term output by the protocol is added to K. The system has a set of rewriting rules on terms in K which model the capabilities of the adversary. Under the Dolev-Yao adversary model [DY83], the adversary can concatenate, de-concatenate, encrypt and decrypt if it has the key. Hence secrecy is usually expressed as a membership constraint on K.

6.2.2 Specifying Authentication

Authentication is a correspondence property. The main goal of authentication is to establish the identities of the principals in the protocol. A correspondence property simply states that one event corresponds to another in a protocol, i.e. it ties 2 protocol events together. Consider a 2-party authentication protocol which authenticates a client C to a server S. The authentication requirement, entity authentication in this case, of this protocol can be stated as "If S accepts credentials and establishes a session with C, C must have initiated communication and sent credentials." Here the correspondence is between the *Accept* event on the server side and an *Initiate* event on the client side.

6.2.3 Key Exchange

Key Exchange can be defined as a correspondence between the *Key Accept* event and *Key Send* event i.e. in a 2-party (A and B) key exchange protocol, if B accepts a key, then it must be the same key sent by A.

6.2.4 Electronic Commerce - related properties

In an electronic commerce protocol such as 1KP, payment authorization can be specified as "If a customer's credit card is debited, the customer must have authorized that debit." Thus payment authorization can be specified as an assertion on protocol events. In this assertion, we may have to reason about the past and future of a protocol state.

6.2.5 Assertions on traces

Other security properties can also be specified as assertions on protocol traces. Anonymity of a principal in a protocol can be specified as "If any principal P knows the real name of principal A, then P always knew the name of A." This is an assertion on the preceding states, at every state on a protocol trace. This assertion may also contain references to the past and the future.

Hence, security requirements of protocols can normally be expressed as correspondence assertions or as constraints on protocol traces. Thus logic is a suitable language to specify security protocol requirements. Some specification mechanisms explicitly use temporal logic for reasoning about the past and future and some others capture that in the semantics of the logical predicates. Using logic to specify security protocol requirements isn't new. We now provide references to the verification literature where this is done. These are in no way exhaustive. Our only purpose is to illustrate that logic is a suitable specification language.

6.2.6 Model Checkers

1. **BRUTUS** by Clarke et al. [CJM98, CJM00, Mar01] is a special-purpose model checker for security protocols. It uses temporal logic to specify security requirements. This temporal logic variant has quantifi ers ranging over the instances of the model; it has atomic propositions that refer to variable bindings, protocol agents and actions and, the past time modal operator so that one can refer to actions that took place in the history of a particular protocol trace. The terms of this logic, i.e. the terms on which the atomic propositions are constructed, can specify cryptographic operations such as encryption and hashing.
2. While BRUTUS is a specialized model checker, Mitchell et al. [MMS97] describe security protocol verifi cation using the general purpose Murϕ model checker [Dil96]. Mitchell et al. [MMS97] use a similar approach for security requirements specifi cation though it does not include a temporal operator.
3. Low [Low96a, Low96b] also uses correspondence assertions to specify authentication properties. Writing correspondence assertions can be subtle. Hence, Lowe [Low98] defi nes high-level fi rst-order predicates which can be used to specify security properties. These predicates are in turn translated into correspondence assertions for analysis.
4. Li [LM05] uses First Order Logic to specify Simple Public Key Infrastructure (SPKI) and Simple Distributed Security Infrastructure (SDSI) Key and Trust Management.

6.2.7 Theorem Provers

In the case of verifi cation by theorem proving, security requirements are theorems which ultimately have to be proved from a set of axioms and deduction rules on protocol actions. They are expressed in logic, typically in fi rst-order logic (FOL). These theorems are similar to the assertions examined earlier.

Paulson [Pau98, Pau97] describes the verifi cation of security protocols, including a complex recursive authentication protocol using the Isabelle theorem prover where security requirements are expressed as theorems to be proved in Higher Order Logic. Paulson has successfully verifi ed more complex protocols as well as in [Pau99] that describes the specifi cation and verifi cation of the TLS (SSL 3.1) protocol and in [Pau01, BMP05, BMPT00, BPM02] that describe the verifi cation of the Secure Electronic Transaction Protocol (SET), one of the cornerstones of E-commerce.

6.2.8 Using the Z Specification Language

It has been mentioned earlier that logic is an effective specifi cation language for security protocols. Z is a popular specifi cation language using sets, relations, functions and logical

assertions.

Long et al. [LFC03] show how to model cryptographic protocols involving most of the common cryptographic operations such as symmetric and asymmetric encryption, message digests, and nonces, in the Z specifi cation language. Through a case study on the Needham-Schroeder public-key protocol, they have shown how a security requirement can be specifi ed as a Z-invariant and how that invariant can be *manually* proved or disproved. The central idea behind Long et al.'s modeling is that protocols involve disjoint types of data, e.g. keys, nonces, timestamps, addresses, encrypted data that can be specifi ed using data types in Z, cryptographic operations that are mathematical functions and be specifi ed as well, and logical constraints [LFC03].

6.3 Access Control (AC) Policy Specification

In this section, we survey visual (graphical) mechanisms used to specify access control policies. A similar analysis has been given by Koch and Parisi-Presicce [KPP03].

6.3.1 Graph-based Specifications

Koch et al. [KMPP01a] describe a graphical method to specify access control policies. This method consists of type graphs and instance graphs. The nodes of the type graph represent system *entity types* and each edge represents a relationship between 2 entity types. Hence each access control policy model is represented by a type graph. For example, *User* and *Role* are 2 nodes in a type graph representing RBAC [SCFY96, SFK00]. There is an edge between *User* and *Role* labeled *belongs to* or *is in*.

Access Control policy models leave the nature of the protected resources open. A particular system's access control policy is obtained by instantiating the policy model. An instance graph G is an instance of a type graph T, if and only if for each node and edge in G, there is a corresponding node and edge type in T. An instance graph is a snapshot of the system's current access control policy.

Access Control policies often evolve over time. Evolution is specifi ed through graph rewriting rules, or graph transformation rules. Rewriting rules are defi ned on type graphs. They may be applied to instance graphs to deduce the current state of the system. Koch and Parisi-Presicce [KPP02b] extend this method to include both positive and negative constraints for declarative specifi cation. Their work also describes how constraints and graph rules can be used to specify the composition of properties. Koch et al. [KMPP01b, KPP02b] describe the formal semantics of this graphical language.

Using a graph-based specifi cation method facilitates verifi cation. An instance graph represents a state of the system. Verifi cation explores the state space where each transition is made possible by a graph rewriting rule. Verifi cation of access control models is usually concerned with generating and exploring this state-space to check for:-

1. Coherence between policy rules and constraints, i.e. whether or not all policy rules construct states satisfying the constraints. This is done through a pairwise verifi cation of Constraint-Constraint, Constraint-Rule and Rule-Rule pairs [KMPP02a], or

2. Safety issues such as whether or not a given subject gets a given permission or not. As shown by Koch et al., in general, safety for graph-based models is undecidable [KMPP02b]. However, they also show that under reasonable restrictions on policy rules, it is decidable.

Koch et al. [KMPP02c, KMPP00] describe the specifi cation and verifi cation of RBAC [SCFY96, SFK00] in this formalism.

6.3.2 UML-based specifications

Koch and Parisi-Presicce [KPP02a] model access copntrol policies in UML using class and object diagrams, stereotypes and OCL constraints. This is done by:-

1. Representing various entities and relationships in the AC policy by a class diagram. For example, entities in Role-Based Access Control are various roles and their assigned permissions, users, sessions and objects.

2. Representing policy specifi c rules through UML stereotypes.

3. Special constraints, as in programmatic access control, through OCL.

It appears that the approach taken by Koch and Parisi-Presicce [KPP02a] is generic and can specify most access control policies though there is no proof of its expressivity. Koch and Parisi-Presicce [KPP03] provide an example specifi cation of an RBAC policy and [KPP02a] contains an example specifi cation of View-based Access Control (VBAC), which is an extension of RBAC for distributed object-oriented systems.

Also, the verifi cation of UML-based access control policy specifi cations is done by translating them into the graphical framework mentioned in Section 6.3.1. Koch and Parisi-Presicce [KPP02a] describe this translation procedure.

Park and Kwon [PK05] model access control policies through UML and use this model of the system and access control policy to generate Alloy Specifi cation [Jac05] which is then analyzed through the Alloy analyzer. They also employ the USE tool [Ric] to analyze parts of the UML specifi cation.

Alloy Specifi cation of RBAC: Schaad and Moffet [SM02] use the Alloy Specifi cation Language to specify RBAC. Alloy seems to be the most suitable specifi cation language for Access Control policies because of its modularity and a conventional programming language-like syntax. Specifi cation in Alloy has the added advantage that it can be verifi ed by the

Alloy analyzer. Schaad and Moffett [SM02] have formally specified RBAC96, ARBAC97, i.e. URA97, PRA97, RRA97 and Separation of Duty constraints in addition to analyzing conflicts between SoD and ARBAC97 using the Alloy analyzer. RBAC96 was the initial RBAC model as proposed by Sandhu et al. [SCFY96]. RBACs motivation is to simplify administration of authorizations. ARBAC97 or Administrative-RBAC uses RBAC itself to manage RBAC, i.e. to have administrative roles in RBAC. This provides administrative convenience and scalability, especially in decentralizing administrative authority, responsibility, and chores. ARBAC97 [SBM99] has 3 components namely URA97 (user-role assignment 97), PRA97 (permission-role assignment 97), and RRA97 (role-role assignment 97) dealing with different aspects of RBAC administration. For more information on Separation of Duty, see Sandhu et al. [SCFY96], Li et al. [LTB05], Simon and Zurko [SZ97].

Ponder Specification Language: Ponder [DDLS01, LSDD00] is a specialized Policy Specification Language with a programming-language like syntax. It supports the specification of most access control concepts and constructs, delegation, obligation constraints and authorization constraints. Ponder specifications can be written in a structured manner and can be grouped into composite specifications which are essential in large information systems and organizations. Ponder is not specific to information security. It is a generic policy specification language.

6.3.3 SecureUML

UMLSec stereotypes cannot be used to specify access control policies. Basin et al. [LBD02, BDL04] have designed another UML profile SecureUML to model and specify access control policies of distributed systems. SecureUML supports the specification of declarative access control mechanisms, or access control policies, and programmatic access control mechanisms, business logic to override the default access control policy. Declarative mechanisms are specified using the UML Profile and programmatic mechanisms using UML's Object Constraint Language (OCL).

SecureUML cannot model and specify protected resources. RBAC being a generic security mechanism that leaves open the nature of the protected resources, SecureUML does not have stereotypes for system modeling. Basin et al. [BDL04] define UML profiles for:-

1. **Distributed Object Oriented Systems - ComponentUML:** Basin et al. [BDL04] show how to combine SecureUML and ComponentUML. They define a SecureUML version with ComponentUML as the system design language. These SecureUML models are translated into Enterprise Java Bean (EJB) components and deployment descriptors which specify the access control properties of those components. The access control model of EJB is based on Role-based Access Control (RBAC). The security subsystem of EJB Application Server is responsible for implementing access

control policy as specifi ed in the deployment descriptors on components. In addition to all this, programmatic access control specifi cations are translated into Java assertions in the Bean class.

2. **Multi-tiered Applications - ControllerUML:** ControllerUML is based on state machines. The idea behind this profi le is that in multi-tier applications, usually developed in accordance with the Model-View Controller pattern [GVJH98], the controller manages the data fbw between different views. Its behavior can be modeled by state machines and there are access control issues w.r.t. views accessing the data sets. Basin et al. [BDL04] also show how to combine SecureUML and ControllerUML and transform these models into Java Servelet Components with deployment descriptors. The execution environment supports RBAC.

Both these frameworks realize Model-driven security *partially*. SecureUML models are not formally verifi ed. Verifi cation and Test Generation mechanisms for SecureUML are prospective areas of research.

6.3.4 Other Access Control Specification Methods

Gavrila and Barkley [GB98] use set theory, functions and fi rst-order logic to specify RBAC96, URA97 and RRA97. Such specifi cation is feasible because *Users, Roles, Permissions* are sets, assignment of users to roles and permissions to roles are functions, role hierarchies are relations, some are partial orders, and constraints are logical statements with fi rst-order quantifi cations. Crampton [Cra03] specifi es constraints, including Sod, in RBAC through set theory and propositional logic. The difference between Gavrila and Barkley's [GB98] and Crampton's [Cra03] work is merely syntactic. Bertino et al. [JBG05, JBLG05] uses logic to give a formal semantics for the Generalized Temporal RBAC model (GTRBAC) which can be leveraged to specify TRBAC policies.

6.4 Requirements Engineering for Software Security

Security is considered explicitly in both the SDLs during the Requirements Engineering phase. Desired security requirements of the software should be explicitly specifi ed.

Microsoft advocates using a separate security assessment team to engineer and evaluate the security of its products. It is the responsibility of the software development team to identify all functional requirements including the security functional requirements. Each team has a security engineer who reviews the product plan, functional requirements and determines security milestones and exit criteria. These requirements are well-documented.

McGraw's SDL advocates the use of Abuse Cases in the Requirements Engineering phase.

7 Analysis and Design

7.1 UMLSec models in analysis and Design

UMLSec includes all UML Analysis and Design artifacts like activity diagrams, deployment diagrams, sequence diagrams and statecharts. The design of the system can thus be represented in UMLSec. These diagrams can use UMLSec stereotypes and can have constraints associated. That is how security requirements are integrated with system design in UMLSec.

Jan Jürjens and colleagues have developed a set of tools to analyze UMLSec models [JÖ5]. These models can be stored in an XMI file format and analyzed by a static as well as a dynamic analyzer. The static analyzer, as its name implies, analyzes the static features of UMLSec models. The dynamic analyzer is either a model checker or a theorem prover.

Jürjens and Shabalin [JS04] describe the SPIN model checker. Here, the UMLSec models are transformed into PROMELA - the modeling language of the SPIN model checker and security requirements are transformed into *never-claim* or *should not happen* assertions in PROMELA code. The security requirements on the UMLSec models are then verified by model checking. This approach suffers from the state-space explosion problem just as with any other model checker and leads to undecidable problems for infinite state systems.

Jürjens [JÖ5] and Jürjens and Shabalin [JS05] describe the use of the (ATP) SETHEO theorem prover for first order logic (FOL). Consequently, UMLSec models are translated into FOL. The results from this formal verification step are fed back into the Error Analayzer which reports them to the user along with those of the static analyzer. This approach has been used in the development of:-

1. A Security Centric Biometric Authentication System from T-Systems Inc. and Project Verisoft [JÖ5]
2. An attack on the TLS protocol [JÖ5]
3. Smart-card based Electronic Purse Specification for VISA International and Oktoberfest [Jür04, JÖ1].
4. Multi-layer security protocol for HypoVereinsbank web application [HJ03]
5. Telemedicine Applications [Jür02]

As already mentioned, UMLSec realizes the Model-driven security paradigm only partially. Also, research and case studies have focussed on using UMLSec to model cryptographic systems and specify their properties such as secrecy, authentication, integrity,

non-repudiation, fair exchange, electronic commerce systems and information flow. Generation of code or application infrastructure and of test cases, their adequacy, etc. are open questions and areas of prospective research.

7.2 Analysis phase in Software Security - Life Cycles

In both the SDLs, the focus of the analysis phase is risk identification and assessment. In Microsoft's SDL, the primary software-security tasks during requirement analysis and design is threat modeling and response. Threat modeling [SS04] consists of the following steps:-

1. Decomposing the application using requirement specifications, UML activity diagrams, data flow diagrams and other UML analysis-phase artifacts to identify the systems' boundaries, trusted and un-trusted components, and data/resources (targets) that need protection.

2. Identifying threats against each system target from the decomposition process *manually*. This is done by training software engineers on common security vulnerabilities. A framework based on common security vulnerabilities (STRIDE - Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege) is used to manually identify threats against targets. The team attempts to identify such threats at the network, host and application levels.

3. Each threat is documented and modeled using attack trees which are decision trees depicting the path to compromise. They provide a hierarchical, precise and structured description of attacks.

4. The threats are prioritized because it may not be worthwhile to address all of them. Microsoft uses the DREAD model which assesses threat based on its Damage Potential, Reproducibility, Exploitability, Affected Users and Discoverability to focus on threats to be mitigated.

   Risk can be defined as the probability that a given threat actually occurs. Microsoft's SDL focusses on technical risk analysis based on analysis and design artifact using specific methodologies. In contrast, McGraw's SDL [VM04] takes a broader approach. It advocates risk analysis based on requirements and business process/context. Thus Risk Identification is a manual process based on a set of guidelines mined from existing security vulnerabilities. However, there does not exist a standard assessment method for software security risks. See [VM04] for a survey of various risk measurement (quantification) techniques for software security.

In both the SDLs,

1. Based on the risk associated, system engineers identify threats which have to be addressed by the software system
2. System Engineers respond to threats by using security mechanisms such as authentication, access control, digital signatures, security design patterns etc. In other words, they try to mitigate identifi ed risks.

Swiderski and Snyder's book on threat modeling [SS04] provide a detailed description of the processes mentioned above.

The completed design is subjected to a **security review** which focusses on refuting identifi ed threats. This is a manual review.

8 Implementation

It is advocated that the main focus during coding should be to prevent common (a.k.a known) security-related flaws in code. So, Microsoft and McGraw's SDLs recommend using secure programming practices. Microsoft has published a set of best practices in the book [HL02, HM04].

Once coding is complete, static analysis is used to detect common implementation errors. Tools used by Microsoft in its SDL include PREfi x, PREfast, PREsharp and espX. These tools have a common functionality but differ in the techniques used and the target language. For example, PREfi x tool detects defects in C and C++ code using symbolic analysis. Knowledge gained from security exploits is fed back into the static analyzer i.e. used to develop the static analyzer and all further code is statically analyzed.

Static analysis tools seem to perform better than manual audits because they are faster and encapsulate security knowledge in such a way that does not require the operator to have security expertise. The focus in static analysis is to *Aim for good, not perfect*. The output of the static analyzer still requires human intervention to rule out false positives.

A static analyzer is considered sound if for a given set of assumptions, it may produce false positives but does not let a false negative slip by. Chess and McGraw provide a survey of static analysis tools [CM04b].

9 Testing

9.1 Software Security Testing

The key issue in software security testing is how to test for un-specifi ed behavior. Normally, in conformance testing and specifi cation-based testing, the focus is to look for the presence of some *correct* behavior and not the *absence* of additional behavior. This additional behavior may be manifested during the execution against a test input test case but

may not be noticeable. For example, a tester while applying input X to an application may look for result Y, but may not notice a trojan horse in the application which opens an FTP connection to a server or the creation of a temporary fi le with confi dential information.

In our opinion, signifi cant research is needed to address the issue mentioned above. Software security testing has traditionally focussed on constructing test cases from known exploits. The idea is to maintain an exploit library and check if these exploits still work on new software products. Or, to use previous exploits as templates and try to construct test cases based on them. This is not the best testing technique, but is nevertheless important because of the proliferation of free attack tools on the internet.

Many attackers are novice computer users who like to *play around* with systems. It is widely acknowledged that such attackers use *published* attack patterns. Also, past exploits give us a good idea of the attackers' approaches which software engineers use to generate tests *thinking like what the attackers thought*. See [Tho03, DWW99] for an overview.

Testing based on published attack patterns offers little defense against determined hackers. One approach to detecting unspecifi ed behavior during testing is to use program monitors. Examples are the Regmon (For Registry Monitor) and FileMon (File Monitor) tools produced by SysInternals (www.sysinternals.com) for monitoring the registry and fi le system of Microsoft Windows, App-sight - a tool to monitor environmental interactions produced by Identify Software (www.identify.com).

In both the Security Development Lifcycles (SDLs) discussed earlier, Figures 3 and 2, testing consists of 2 stages. One stage is testing the security functionality using standard functional testing and the other of using risk assessments, attack patterns and threat models to manually derive tests. Each threat that is addressed by the design corresponds to at least one test case in the test set.

The focus in risk-based security testing is to act like the attacker. McGraw [MP04] advocates training testers on security vulnerabilities and attack patterns so that they are able to construct effective test cases. There have been limited attempts by McGraw and Potter [MP04] describe an approach at Cigital Corp. to automate software security testing for Java Cards. They use the above mentioned approach to automatically generate test cases. It is possible to use this approach since it is specifi c to a language and the application set.

McGraw [McG03] also employed intelligent fuzzing or data-mutation based software testing. This involved deriving application interfaces from the design and making sure that code correctly handles all data entering the interface. Test cases are derived by constructing input data that violate the API rules and using different mutations of the same. McGraw [HMA04] recommends that abuse cases be used to derive tests. Such interface-based mutation approach was proposed earlier by Maldonado et al. [DMM01] and Ghosh et al [GM01] in the context of testing large applications. McGraw's idea is that the software system must ensure that any given abuse case is not manifested in it. Test plans are derived from the design, in parallel with the coding phase. Test adequacy is determined

through an external review of the security test plan.

9.2 Security Functional Testing

Security Software needs to be tested. There are several implementations of security protocols which have to be tested for conformance. Model-based testing or specifi cation-based testing seems to be a promising approach to test for security conformance. As mentioned earlier, security protocols, access control schemes can be formally specifi ed in UML, Alloy, Z, B and a host of other languages. There has been a signifi cant amount of research on generating test cases from formal specifi cations. More research is needed on the suitability of the model-driven testing approach to security. A survey of model-based testing techniques is beyond the scope of this report. We give pointers in [MC99, SC96, SCS97, HNS97, Pre05] to the appropriate literature. Again, a comprehensive list of pointers appears to be too long. So, we refer the reader to the book [BJKP05]

9.3 Testing Firewalls

A fi rewall is a useful mechanism to protect the internal network of an organization. Firewalls implement the network access policy if an organization. Some typical components of a fi rewall are Stateful Packet Filter –which blindly looks at the types of packets and their sequences, Content Filter–which inspects transferred data irrespective of the conveying protocol, Network Address Translator, Application level gateway and Event Logger. It is interesting to think of fi rewall rules and specifi cations in terms of traditional access control models though more research is needed to rigorously relate them.

Senn, Basin and Caronni [SBC05] describe the specifi cation based testing of deployed fi rewalls. They also describe the syntax and semantics of a simple fi rewall policy description language. A fi rewall policy formalizes the various types of traffi c allowed between different zones, a zone being a portion of the network separated from the rest of the network by fi rewalls. The formal fi rewall specifi cation is converted into a Mealy automaton and abstract test cases are generated using the Unique Input/Output Sequences (UIO) method [SD88]. The abstract test cases are then instantiated for various protocols and IP addresses which are present in the policy specifi cation. They assume that fi rewalls are stateful packet fi lers and are thus able to keep the specifi cation language simple and convert the spec into a deterministic FSM. The disadvantage is that complex fi rewalls which use timing and sequence numbers cannot be modeled. But, this approach of testing a deployed fi rewall not only helps uncover errors in its implementation but also in its confi guration.

Jürjens and Wimmel [JW01] describe the specifi cation-based testing of stateful packet fi lters using a CASE tool. The fi rewall is specifi ed in AUTOFOCUS, a tool for graphically specifying distributed systems [HSSS96, SPHP02]. The structural view of the network

is described by a system structure diagram (SSD) and each component of the SSD is specifi ed by a State Transition Diagram (STD). STDs are Extended Finite State Machines (EFSMs).

Jalili and Rezvani [JR02] also describe a simple specifi cation language for stateful packet fi lters akin to that of Senn et al. [SBC05]. Jalili and Rezvani do not derive test cases but formally verify the consistency of the fi rewall specifi cation using a theorem prover. More research is needed to verify the correctness and reliability of the fi rewall specifi cation.

It is recommended that fi rewalls be also subjected to vulnerability-based testing i.e. deriving test cases from known fi rewall vulnerabilities (see Fahmy et al. [KFS+03] for a description of some fi rewall vulnerabilites).

9.4 Testing Intrusion-detection systems

As we know, Intrusion Detection Systems help identify intrusions or misuse of computer systems by either authorized users or external perpetrators. IDSs detect intrusions by analyzing information about user activity from sources such as system tables, audit records, logs and network traffi c summaries. IDSs heavily depend on information about past exploits to detect intrusions. The quality of an IDS is also heavily influenced by the quality of its *intrusion signatures* or intrusion models. A perfect model would be able to detect all instances of the modeled attack without making mistakes. In technical terms, a perfect model would produce a 100% detection rate without false positives.

So, how exactly can one test intrusion detection systems ? Testing IDSs is done in two stages— one by the manufacturer who tests the IDS for conformance i.e. whether the IDS is able to get a 100% detection rate for its model. We could not fi nd any literature on the specifi cation-based testing of IDSs.

The next stage in IDS testing is at the site of deployment. The customer must test the effectiveness of the IDS. Manufacturers do not provide guarantees for the effectiveness of any IDS. In fact, in most cases, the *intrusion signatures* of the IDS are not published or even shared with the customer. This is because, if the *intrusion signatures* are known, an attacker could easily generate attack scripts to circumvent the IDS. Normally, once an IDS is manufactured, it is tested and benchmarked based on its ability to detect a particular suite of intrusions. This *test suite* is obtained manually by *mining* past exploits. DARPA sponsored an IDS testbed at the Lincoln Laboratory of MIT [LHF+00a, LHF+00b]. Though there have been critiques of the effectiveness of its test suite [McH00], this is the fi rst effort in benchmarking based on black-box testing of IDSs.

Vigna, Robertson, and Balzarotti [VRB04] describe a technique to test IDSs using mutations. Note that this black box testing technique is different from mutation testing, which is a white-box technique to evaluate the effectiveness of a test suite. Vigna et al. apply mutant operators to attack templates to automatically generate a large number of variations of an attack which are used as test cases to test the IDS. Obviously, this does not provide a

formal evaluation of the goodness of the detection model, but nevertheless it is an effective technique to increase one's confi dence in the generality of an IDS's model.

Puketza et al. [PZC+96] advocate using equivalence partitioning for the IDS test case selection problem. They recommend that published intrusions be grouped into equivalence classes based on their intrusion signatures and software vulnerabilities exploited. Hence we get two equivalence partitioning schemes and two test suites, one corresponding to each scheme.

10 Summary and Conclusion

In this report, we have surveyed the state of the art in *Software Engineering for Security*. All work surveyed here is state of the art though some, such as the SDLs in Figures 2 and 3 are state of the art as well as *state of the practice*.

Software Security relies heavily on best practices. Knowledge gained from attacks is mined to model threats, analyze risk associated with each of them and mitigate them in the software lifecycle. More research is needed on the use of formal methods for software security. This includes research on using formal specifi cations for software vulnerabilities, threats and automated generation of test cases from specifi cations. The ideal scenario would be to use formal methods and remove manual intervention in every phase of the lifecycle, but we believe that further research is needed for that. Some research effort has gone into the classifi cation of security vulnerabilities (see for example Aslam et al. [AKS96]). It remains to be seen how such classifi cation can be leveraged during design and testing for security.

In Network Security and Access Control, signifi cant research has been reported on formal specifi cation of the system and its requirements. As we mentioned earlier, most of this research is in the context of Formal Verifi cation of Security Protocols. More research is needed on the integration of security requirements with that of the system and the generation of code from the design. Also, more research is needed on specifi cation techniques for complex protocols, e.g. the Data Integrity Protocol for Distributed Media Streaming [HXA+05] and Group Key Agreement protocols [AKNR+04].

There has been a lot of research on specifying security protocols and access control mechanisms in formal languages. Also, there has been wealth of research on Formal specifi cation based testing. Exploring the transitivity here–Specifi cation-based testing for security and its effectiveness, is an interesting area of research. One of the key questions to be answered is whether existing test generation algorithms are suitable for security testing. Another issue is whether security testing should focus on functional testing or whether it should, in addition, include formal risk-based and threat-based testing.

There has always been a debate on whether security requirements should be treated as functional or non-functional requirements. The main purpose of certain software is to

ensure communication security, access control, etc. which are functionalities. So it is argued that security requirements should be functional. On the other hand, security is an emergent feature of a system, i.e. it is a product of the effectiveness of the software engineering practices adopted and also, new ways to attack a system emerge everyday. So, should security be a non-functional requirement ? This difference plays a critical role in the importance assigned to security requirements by the software engineering team. It is recommended that security be always treated as a high-priority requirement and not as an add-on. We may never answer the question whether security is a functional or a non-functional requirement but, the idea is that it should not matter as long as we assign great importance to it.

Research on security testing has not focussed on unit testing to the best of our knowledge. A lot of research is needed on engineering fi rewalls, intrusion detection systems and obfuscation software.

Acknowledgements

Thanks to Scott Miller for his many suggestions and explanations that have gone into making this report more complete than it would be otherwise.

References

[AKNR+04] Yair Amir, Yongdae Kim, Cristina Nita-Rotaru, John L. Schultz, Jonathan Robert Stanton, and Gene Tsudik. Secure group communication using robust contributory key agreement. *IEEE Trans. Parallel Distrib. Syst.*, 15(5):468–480, 2004.

[AKS96] Taimur Aslam, Ivan Krsul, and Eugene H. Spafford. A taxonomy of security vulnerabilities. In *19th National Information Systems Security Conference*, pages 551–560, October 1996.

[Ale02] Ian F. Alexander. Initial industrial experience of misuse cases in trade-off analysis. In *RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 61–70, Washington, DC, USA, 2002. IEEE Computer Society.

[Ale03] Ian F. Alexander. Misuse cases help to elicit non-functional requirements. *Computing & Control Engineering Journal*, 14(1):40–45, 2003.

[Ale04] Ian F. Alexander. Misuse cases: Use cases with hostile intent. *IEEE Software*, 02(3):90–92, 2004.

[Bal05] Thomas Ball. The verifi ed software challenge: A call for a holistic approach to reliability. In *Verified Software: Theories, Tools, Experiments*. to appear, October 2005.

[BDL04] David Basin, Jürjen Doser, and Toorsten Lodderstedt. Model-driven security. *Engineering Theories of Software Intensive Systems: Marktobendorf Summer School*, 2004.

[BJKP05] Manfred Broy, Bengt Jonsson, Joost Pieter Katoen, and Alexander Pretschenr, editors. *Model-based Testing of Reactive Systems: Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[BMP05] Giampaolo Bella, Fabio Massacci, and Lawrence C. Paulson. An overview of the verifi cation of set. *Int. J. Inf. Sec.*, 4(1-2):17–28, 2005.

[BMPT00] Giampaolo Bella, Fabio Massacci, Lawrence C. Paulson, and Piero Tramontano. Formal verifi cation of cardholder registration in set. In *ESORICS '00: Proceedings of the 6th European Symposium on Research in Computer Security*, pages 159–174, London, UK, 2000. Springer-Verlag.

[BPM02] Giampaolo Bella, Lawrence C. Paulson, and Fabio Massacci. The verifi cation of an industrial payment protocol: the set purchase phase. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 12–20, New York, NY, USA, 2002. ACM Press.

[CA01] Hoi Chang and Mikhail J. Atallah. Protecting software code by guards. In *Digital Rights Management Workshop*, pages 160–175, 2001.

[CILN02] Robert Crook, Darrel C. Ince, Luncheng Lin, and Bashar Nuseibeh. Security requirements engineering: When anti-requirements hit the fan. In *RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 203–205, Washington, DC, USA, 2002. IEEE Computer Society.

[CJM98] Edmund M. Clarke, Somesh Jha, and Wilfredo R. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *PROCOMET '98: Proceedings of the IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods*, pages 87–106, London, UK, UK, 1998. Chapman & Hall, Ltd.

[CJM00] Edmund M. Clarke, Somesh Jha, and Wilfredo R. Marrero. Partial order reductions for security protocol verifi cation. In *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 503–518, London, UK, 2000. Springer-Verlag.

[CM04a] Brian Chess and Gary McGraw. Software security. *IEEE Security and Privacy Magazine*, 2(2):53–84, 2004.

[CM04b] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6):76–79, 2004.

[Cra03] Jason Crampton. Specifying and enforcing constraints in role-based access control. In *SACMAT '03: Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 43–50, New York, NY, USA, 2003. ACM Press.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Arizona, July 1997.

[CTL98a] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *ICCL '98: Proceedings of the 1998 International Conference on Computer Languages*, page 28, Washington, DC, USA, 1998. IEEE Computer Society.

[CTL98b] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196, New York, NY, USA, 1998. ACM Press.

[DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specifi cation language. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, London, UK, 2001. Springer-Verlag.

[Dil96] David L. Dill. The Murϕ verifi cation system. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 390–393, London, UK, 1996. Springer-Verlag.

[DMM01] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: an approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, March 2001.

[DWW99] Alden Dima, John Wack, and Shukri Wakid. Raising the bar on software security testing. *IT Professional*, 1(3):27–32, 1999.

[DY83] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[GB98] Serban I. Gavrila and John F. Barkley. Formal specifi cation for role based access control user/role and role/role relationship management. In *RBAC '98: Proceedings of the third ACM workshop on Role-based access control*, pages 81–90, New York, NY, USA, 1998. ACM Press.

[GM01] S. Ghosh and A. P. Mathur. Interface mutation. *Journal of Testing, Verification and Reliability*, 11(4):227–247, December 2001.

[GVJH98] Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. *Design Patterns CD: Elements of Reusable Object-Oriented Software, (CD-ROM)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[HJ03] Siv Hilde Houmb and Jan Jürjens. Developing secure networked web-based systems using model-based risk assessment and UMLsec. In *APSEC*, pages 488–, 2003.

[HL02] Michael Howard and David E. Leblanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2002.

[HL03] Michael Howard and Steve Lipner. Inside the windows security push. *IEEE Security and Privacy*, 1(1):57–61, 2003.

[HM04] Greg Hoglund and Gary McGraw. *Exploiting Software: How to Break Code*. Pearson Higher Education, 2004.

[HMA04] Paco Hope, Gary McGraw, and Annie I. Antn. Misuse and abuse cases: Getting past the positive. *IEEE Security and Privacy*, 2(3):90–92, 2004.

[HNS97] Steffen Helke, Thomas Neustupny, and Thomas Santen. Automating test case generation from z specifi cations with isabelle. In *ZUM '97: Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation*, pages 52–71, London, UK, 1997. Springer-Verlag.

[HSSS96] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. Autofocus: A tool for distributed systems specifi cation. In *FTRTFT '96: Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 467–470, London, UK, 1996. Springer-Verlag.

[HXA+05] Ahsan Habib, Dongyan Xu, Mikhail J. Atallah, Bharat K. Bhargava, and John Chuang. A tree-based forward digest protocol to verify data integrity in distributed media streaming. *IEEE Trans. Knowl. Data Eng.*, 17(7):1010–1014, 2005.

[JÖ1] Jan Jürjens. Modelling audit security for smart-card payment schemes with UML-SEC. In *Sec '01: Proceedings of the 16th international conference on Information security: Trusted information*, pages 93–107, 2001.

[JÖ2] Jan Jürjens. Umlsec: Extending UML for secure systems development. In *UML '02: Proceedings of the 5th International Conference on The Unifi ed Modeling Language*, pages 412–425, London, UK, 2002. Springer-Verlag.

[JÖ3] Jan Jürjens. *Secure Systems Development with UML*. SpringerVerlag, 2003.

[JÖ5] Jan Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 322–331, New York, NY, USA, 2005. ACM Press.

[Jac05] Daniel Jackson. The Alloy specifi cation language and analyzer. In *http://alloy.mit.edu*, 2005.

[JBG05] James B. D. Joshi, Elisa Bertino, and Arif Ghafoor. An analysis of expressiveness and design issues for the generalized temporal role-based access control model. *IEEE Trans. Dependable Secur. Comput.*, 2(2):157–175, 2005.

[JBLG05] James Joshi, Elisa Bertino, Usman Latif, and Arif Ghafoor. A generalized temporal role-based access control model. *IEEE Trans. Knowl. Data Eng.*, 17(1):4–23, 2005.

[JR02] Rasool Jalili and Mohsen Rezvani. Specifi cation and verifi cation of security policies in fi rewalls. In *EurAsia-ICT '02: Proceedings of the First EurAsian Conference on Information and Communication Technology*, pages 154–163, London, UK, 2002. Springer-Verlag.

[JS04] Jan Jürjens and Pasha Shabalin. Automated verifi cation of UMLsec models for security requirements. In *UML*, pages 365–379, 2004.

[JS05] Jan Jürjens and Pasha Shabalin. Tools for secure systems development with UML: Security analysis with atps. In *FASE*, pages 305–309, 2005.

[Jür02] Jan Jürjens. Secure systems development with UML - applications to telemedicine. In *International Conference on Telemedicine (ICT)*, 2002.

[Jür04] Jan Jürjens. Developing high-assurance secure systems with UML: A smartcard-based purchase protocol. In *HASE*, pages 231–240, 2004.

[JW01] Jan Jürjens and Guido Wimmel. Specifi cation-based testing of fi rewalls. In *PSI '02: Revised Papers from the 4th International Andrei Ershov Memorial*

Conference on Perspectives of System Informatics, pages 308–316, London, UK, 2001. Springer-Verlag.

[KFS+03] Seny Kamara, Sonia Fahmy, Eugene Schultz, Florian Kerschbaum, and Michael Frantzen. Analysis of vulnerabilities in internet fi rewalls. *Computers and Security*, 22(3):214–232, April 2003.

[KMPP00] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. A formal model for role-based access control using graph transformation. In *ESORICS '00: Proceedings of the 6th European Symposium on Research in Computer Security*, pages 122–139, London, UK, 2000. Springer-Verlag.

[KMPP01a] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. Foundations for a graph-based approach to the specifi cation of access control policies. In *FoSSaCS '01: Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures*, pages 287–302, London, UK, 2001. Springer-Verlag.

[KMPP01b] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. Foundations for a graph-based approach to the specifi cation of access control policies. In *FoSSaCS '01: Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures*, pages 287–302, London, UK, 2001. Springer-Verlag.

[KMPP02a] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. Conflict detection and resolution in access control policy specifi cations. In *FoSSaCS '02: Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, pages 223–237, London, UK, 2002. Springer-Verlag.

[KMPP02b] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. Decidability of safety in graph-based models for access control. In *ESORICS '02: Proceedings of the 7th European Symposium on Research in Computer Security*, pages 229–243, London, UK, 2002. Springer-Verlag.

[KMPP02c] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. A graph-based formalism for RBAC. *ACM Trans. Inf. Syst. Secur.*, 5(3):332–365, 2002.

[KPP02a] Manuel Koch and Francesco Parisi-Presicce. Access control policy specifi cation in UML. In *Critical System Development with UML*, pages 63–78, 2002.

[KPP02b] Manuel Koch and Francesco Parisi-Presicce. Describing policies with graph constraints and rules. In *ICGT*, pages 223–238, 2002.

[KPP03] Manuel Koch and Francesco Parisi-Presicce. Visual specifi cations of policies and their verifi cation. In *FASE*, pages 278–293, 2003.

[LBD02] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. Secureuml: A UML-based modeling language for model-driven security. In *UML*, pages 426–441, 2002.

[LFC03] Benjamin W. Long, Colin J. Fidge, and Antonio Cerone. A Z based approach to verifying security protocols. In *ICFEM*, pages 375–395, 2003.

[LHF+00a] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das. The 1999 DARPA off-line intrusion detection evaluation. *Comput. Networks*, 34(4):579–595, 2000.

[LHF+00b] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das. Analysis and results of the 1999 DARPA off-line intrusion detection evaluation. In *RAID '00: Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, pages 162–182, London, UK, 2000. Springer-Verlag.

[LM05] Ninghui Li and John C. Mitchell. Understanding spki/sdsi using fi rst-order logic. *International Journal of Information Securit*, to appear, 2005.

[Low96a] Gavin Lowe. Breaking and fi xing the needham-schroeder public-key protocol using FDR. In *TACAs '96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166, London, UK, 1996. Springer-Verlag.

[Low96b] Gavin Lowe. Some new attacks upon security protocols. In *CSFW*, pages 162–169, 1996.

[Low98] Gavin Lowe. Casper: a compiler for the analysis of security protocols. *J. Comput. Secur.*, 6(1-2):53–84, 1998.

[LSDD00] Emil Lupu, Morris Sloman, Naranker Dulay, and Nicodemos Damianou. Ponder: Realising enterprise viewpoint concepts. In *EDOC '00: Proceedings of the 4th International conference on Enterprise Distributed Object Computing*, pages 66–75, Washington, DC, USA, 2000. IEEE Computer Society.

[LTB05] Ninghui Li, Mahesh V. Tripunitara, and Ziad Bizri. On mutually-exclusive roles and separation of duty. *Journal of the ACM*, submitted, 2005.

[Mar01] Wilfredo Rogelio Marrero. *Brutus: a model checker for security protocols*. PhD thesis, Carnegie Mellon University, 2001. Chair-Edmund M. Clarke.

[MC99] Ian MacColl and David A. Carrington. A model of specifi cation-based testing of interactive systems. In *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II*, page 1862, London, UK, 1999. Springer-Verlag.

[McG03] Gary McGraw. From the ground up: The dimacs software security workshop. *IEEE Security and Privacy*, 1(2):59–66, 2003.

[McH00] John McHugh. Testing intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by lincoln laboratory. *ACM Trans. Inf. Syst. Secur.*, 3(4):262–294, 2000.

[Mea03a] Catherine Meadows. Formal methods for cryptographic protocol analysis: Emerging issues and trends. *IEEE Journal on Selected Areas in Communications*, 21(1):44–54, January 2003.

[Mea03b] Catherine Meadows. What makes a cryptographic protocol secure? the evolution of requirements specifi cation in formal cryptographic protocol analysis. In *ESOP*, pages 10–21, 2003.

[MF99] John McDermott and Chris Fox. Using abuse case models for security requirements analysis. In *ACSAC '99: Proceedings of the 15th Annual Computer Security Applications Conference*, pages 55–65, Washington, DC, USA, 1999. IEEE Computer Society.

[MMS97] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murϕ. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 141–151, Washington, DC, USA, 1997. IEEE Computer Society.

[MP04] Gary McGraw and Bruce Potter. Software security testing. *IEEE Security and Privacy*, 2(5):81–85, 2004.

[Pau97] Lawrence C. Paulson. Mechanized proofs for a recursive authentication protocol. In *CSFW*, pages 84–95, 1997.

[Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Comput. Secur.*, 6(1-2):85–128, 1998.

[Pau99] Lawrence C. Paulson. Inductive analysis of the internet protocol tls. *ACM Trans. Inf. Syst. Secur.*, 2(3):332–351, 1999.

[Pau01] Lawrence C. Paulson. Set cardholder registration: The secrecy proofs. In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, pages 5–12, London, UK, 2001. Springer-Verlag.

[PK05] Sachoun Park and Gihwon Kwon. Verifi cation of UML-based security policy model. In *International Conference on Computational Science and its Applications (ICCSA)*, pages 973–982, 2005.

[Pre05] Alexander Pretschner. Model-based testing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 722–723, New York, NY, USA, 2005. ACM Press.

[PZC+96] Nicholas J. Puketza, Kui Zhang, Mandy Chung, Biswanath Mukherjee, and Ronald A. Olsson. A methodology for testing intrusion detection systems. *IEEE Trans. Softw. Eng.*, 22(10):719–729, 1996.

[Ric] Mark Richters. http://www.db.informatik.uni-bremen.de/projects/USE/.

[SBC05] Diana Senn, David Basin, and Germano Caronni. Firewall conformance testing. In *TestCom*, pages 226–241, 2005.

[SBM99] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur.*, 2(1):105–135, 1999.

[SC96] Phil Stocks and David Carrington. A framework for specifi cation-based testing. *IEEE Trans. Softw. Eng.*, 22(11):777–793, 1996.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[SCS97] Harbhajan Singh, Mirko Conrad, and Sadegh Sadeghipour. Test case design based on z and the classifi cation-tree method. In *ICFEM '97: Proceedings of the 1st International Conference on Formal Engineering Methods*, page 81, Washington, DC, USA, 1997. IEEE Computer Society.

[SD88] Krishan Sabnani and Anton Dahbura. A protocol test generation procedure. *Comput. Netw. ISDN Syst.*, 15(4):285–297, 1988.

[SFK00] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The nist model for role-based access control: towards a unifi ed standard. In *RBAC '00: Proceedings of the fi fth ACM workshop on Role-based access control*, pages 47–63, New York, NY, USA, 2000. ACM Press.

[SM96] Paul F. Syverson and Catherine Meadows. A formal language for cryptographic protocol requirements. *Des. Codes Cryptography*, 7(1-2):27–59, 1996.

[SM02] Andreas Schaad and Jonathan D. Moffett. A lightweight approach to specifi cation and analysis of role-based access control extensions. In *SACMAT*, pages 13–22, 2002.

[SPHP02] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-based development of embedded systems. In *OOIS '02: Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, pages 298–312, London, UK, 2002. Springer-Verlag.

[SS04] Frank Swiderski and Window Snyder. *Threat Modeling*. Microsoft Press, Redmond, WA, USA, 2004.

[SZ97] Richard Simon and Mary Ellen Zurko. Separation of duty in role-based environments. In *CSFW '97: Proceedings of the 10th Computer Security Foundations Workshop (CSFW '97)*, page 183, Washington, DC, USA, 1997. IEEE Computer Society.

[Tho03] Herbert H. Thompson. Why security testing is hard. *IEEE Security and Privacy*, 1(4):83–86, 2003.

[VM04] Denis Verdon and Gary McGraw. Risk analysis in software design. *IEEE Security and Privacy*, 2(4):79–84, 2004.

[VRB04] Giovanni Vigna, William Robertson, and Davide Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *ACM Conference on Computer and Communications Security*, pages 21–30, 2004.