Special issue: The future of software engineering for security and privacy

Survey Paper

A survey on security patterns

Nobukazu YOSHIOKA[1], Hironori WASHIZAKI[2], and Katsuhisa MARUYAMA[3]
[1,2]*National Institute of Informatics*
[2]*The Graduate University for Advanced Studies*
[3]*Department of Computer Science, Ritsumeikan University*

ABSTRACT

Security has become an important topic for many software systems. Security patterns are reusable solutions to security problems. Although many security patterns and techniques for using them have been proposed, it is still difficult to adapt security patterns to each phase of software development. This paper provides a survey of approaches to security patterns. As a result of classifying these approaches, a direction for the integration and future research topics is illustrated.

KEYWORDS

Security, software engineering, design patterns, security patterns, refactoring

1 Introduction

In the face of an increasing number of business services on open networks and distributed platforms, security issues have become critical. Devanbu et al. stated this as follows [1]:

> *"Security concerns must inform every phase of software development, from requirements engineering to design, implementation, testing, and deployment."*

It is difficult to do so, however, because not all software engineers are security specialists and there are many concerns in security. Patterns are reusable packages incorporating expert knowledge. Specifically, a pattern represents a frequently recurring structure, behavior, activity, process, or "thing" during the software development process. A security pattern includes security knowledge and is reusable as a security package.

Recently, many security patterns have been proposed. Nevertheless, it is still difficult to adapt security patterns to each phase of software development. In this paper, we survey current security patterns and offer a classification of them. This then leads us into a number of problems and an indication of potential future research directions for security patterns.

This paper is structured as follows. Section 2 defines security concepts as used in this paper. Section 3 describes various security patterns from the development phase viewpoint. Then, we discuss the problems and potential research directions in section 4. Finally, we conclude this paper in section 5.

2 Background of security

This section defines security concepts and describes security patterns.

2.1 Security concepts

The following are general security concepts and definitions, based on the definitions in [2]:

Asset: Information or resources that have value to an organization or person.

Stakeholder: An organization or person who places a particular value on assets.

Security objective: A statement of intent to counter threats and satisfy identified security needs.

Threat: A potential for harm of an asset.

Attack: An action intended to violate the security of an asset.

Attacker: An entity that carries out attacks.

Vulnerability: A flaw or weakness that could be exploited to breach the security of an asset.

Received October 2, 2007; Revised December 3, 2007; Accepted January 7, 2008.
[1)] *nobukazu@nii.ac.jp*,
[2)] *washizaki@nii.ac.jp*,
[3)] *maru@cs.ritsumei.ac.jp*
DOI: 10.2201/NiiPi.2008.5.5

Countermeasure: An action taken to protect an asset against threats and attacks.

Risk: The probability that a successful attack occurs and the impact of successful violation.

In this paper, we focus on the above security concerns and conduct our survey by paying specific attention to them.

2.2 Security pattern languages

A pattern can be characterized as *"a solution to a problem that arises within a specific context"* [2].

In addition, a pattern includes not only the solution but also the context and problem for which the solution is used. Hilside[1] defines a pattern as follows:

> *"Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves."*

We adapt this concept to security patterns. This means that we do not consider general methods such as SQUARE [3], Tropos [4], UMLsec [5], and so forth.

The following is the general pattern format shown in [2], though it is not the only possible format:

Name, Also Known As, Example,

Context: The situation in which the pattern may apply.

Problem: The problem the pattern addresses, including a discussion of its associated forces.

Solution: The fundamental solution principle underlying the pattern

Structure: A detailed specification of the structural aspects of the pattern, using appropriate notations.

Dynamics: Typical scenarios describing the run-time behavior of the pattern.

Implementation: Guidelines for implementing the pattern.

Example Resolved: Discussion of important aspects for resolving the example.

Variants: A description of variants or specializations of the pattern.

Known Uses: Examples of using the pattern.

Consequences: The benefits that the pattern provides and any potential liabilities.

See Also

In addition, we adapt the following definitions for this paper:

> *A pattern describes both a process and a thing: the 'thing' is created by the 'process'* [6].
>
> *A pattern language is a network of tightly interwoven patterns that defines a process for resolving a set of related, interdependent software development problems systematically* [7].

In general, we can define security patterns as patterns with respect to the security concepts described in section 2.1.

3 Security patterns

In this section, we categorize security patterns from the software lifecycle point of view. Then, we describe security patterns in term of security concepts for each phase of software development. Finally, we discuss engineering achievements and ongoing research on utilizing those security patterns from the pattern lifecycle point of view. Specifically, we show patterns for the requirement phase in section 3.1. Section 3.2 illustrates patterns for the design phase, and then patterns for the implementation phase are described in section 3.3. Section 3.4 describes the achievements and research on utilizing security patterns, including methodologies to develop secure software systems by using patterns.

3.1 Security patterns for requirement phase

In this phase, we first decide what to protect, i.e., the "assets." In addition, we analyze the reasons for protecting these assets in order to decide how to protect them and to what degree. Finally, we specify the security requirements as part of the system requirements.

3.1.1 Analysis process patterns

There are several analysis process patterns for this phase. Security Needs identification for Enterprise Assets is a pattern [2] helps identify assets and illustrates what kinds of information in a system are assets. The pattern includes the identification of business assets, business factors with security influence, the relation between assets and business factors, and the security type for each asset. Fig. 1 illustrates the process.

There are four kinds of security properties [2]:

- *Confidentiality* is the property that data is disclosed only as intended by the enterprise.
- *Integrity* is the property that enterprise assets are not altered in a manner contrary to the enterprise's wishes.
- *Availability* is the property that enterprise assets, including business processes, will be accessible when needed for authorized use.

[1] http://www.hillside.net/patterns/definition.html

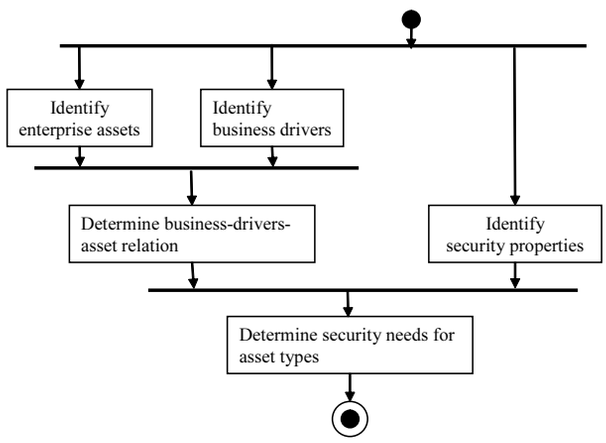


Fig. 1 Security needs identification for enterprise assets.

- *Accountability* is the property that actions affecting enterprise assets can be traced to the actor responsible for the action.

The Asset Valuation Pattern [2] illustrates the guidelines and criteria for asset valuation. Specifically, it defines six degrees of valuation, from negligible to extreme, from three different points of view: security requirements, financial value, and business impact.

It is impossible to develop a completely secure system because of the cost, time, and resources needed for the development, and because of new kinds of attacks. Thus, we must also decide the priority for each asset, which implies a criterion for the threshold of security vs. cost during the development process. The priority of an asset is decided according to not only the value but also the risk for the asset, including likelihood and impact. Therefore, we need to analyze the threat and vulnerability of a system in order to evaluate the risk. Specifically, we use the analysis of the threat to a system as a means of identifying why we protect assets. In addition, the vulnerability of a system is analyzed in order to understand what we are protecting the system from.

The Threat Assessment Pattern [2] is used to reveal the threats to a system. The pattern lists typical threat sources, actions, and consequences. Threat sources can be environmental forces such as earthquakes, deliberate attacks, or accidental errors. To identify and rate system vulnerability, the Vulnerability Assessment Pattern [2] can be used. In addition, the Risk Determination Pattern [2] helps calculate the qualitative rather than quantitative risks for each asset, in order to evaluate the risks relatively.

The Enterprise Security Approaches Pattern [2] helps decide abstract approaches for asset protection. These approaches include prevention, detection of attacks, and response to attacks. The approaches offer a strategy of design, implementation, and maintenance of security functions.

An antipattern describes a pattern that initially appears to be beneficial but ultimately results in bad consequences. Kis proposed antipatterns with respect to security [8]. The solutions for these antipatterns describe situations of successful attacks or bad implementation of security functions.

3.1.2 Model based patterns

There are some patterns using models, such as Tropos or Problem Frame [9], for analysis and specification of security requirements. Tropos [4] is a model for analysis of requirements among stakeholders. A Problem Frame defines an identifiable problem class.

Giorgini et al. proposed patterns specifying a kind of confidentiality by using Secure Tropos in [10]. Secure Tropos is an extension of Tropos that allows for the expression of Ownership, Trust, and Delegation relations. The patterns in [10] are monitoring patterns for when agents transfer personal information. We can use Secure Tropos to formally analyze from whom the permission to handle information should be obtained.

We can model attack situations in which an agent has an attacker's role by using Tropos. Mouratidis et al. proposed patterns for protection against malicious agents by using Tropos [11]. Specifically, they define the Agent Authenticator, Agency Guard, Access Controller, and SandBox patterns.

Hatebur et al. used the Security Problem Frame, based on the Problem Frame, to specify security patterns with security requirements [12]–[14]. The patterns illustrate the requirements specifying that impor-

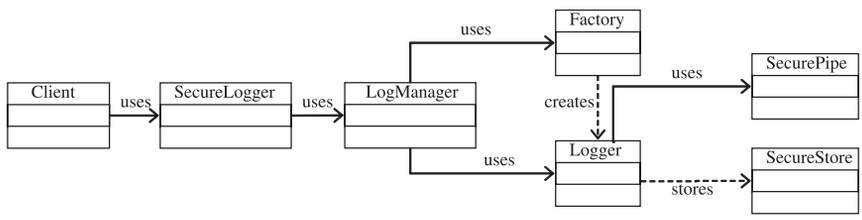


Fig. 2 Secure logger pattern with secure log store strategy.

tant information does not leak to an improper person using a malicious subject frame. In addition, the patterns also specify generic security protocols by using sequence diagrams.

3.2 Security patterns for design phase

This section describes security patterns for decisions on conceptual architecture and the detailed design of systems. In the design phase of software development, we should design security functions to satisfy the security properties of assets identified in the requirement phase. Specifically, we can design such functions using access control, authentication, cryptography, electric signatures, and logging components or services[2)]. What security functions are needed depends on not only the security properties but also the security strategy, such as prevention against attacks or detection of attacks. Against a wiretapping attack, an instance of violating confidentiality, we need to keep logs for detection, in addition to encryption of data for prevention.

Yoder and Barcalow first introduced conceptual security architecture as patterns [15]. They provided a natural language description of seven security patterns: Single Access Point, Check Point, Roles, Session, Full View with Errors, Limited View, and Secure Access Layer Patterns.

Fernandez and Pan illustrated security patterns by using UML diagrams [16] such as that shown in Fig. 2, as well as design patterns [17]. Specifically, the Authorization, Role-Based Access Control, Multilevel Security, and File Authorization patterns are catalogued.

The security patterns book [2] includes 25 total architectural and design-level patterns. In addition, it introduces seven patterns for secure Internet applications.

The following section 3.2.1 describes how to use design patterns with respect to security properties. In section 3.2.2, we introduce some approaches to bridge between design patterns and security requirements. Then, section 3.2.3 shows some domain-specific security patterns.

[2)] We focus on security functions in this paper, although there are other methods of enforcing security properties for assets, such as moral education and enforcement by law or organizational governance.

3.2.1 Design of security properties

Access control models are used to achieve confidentiality of assets in a situation of open networks. In [2], five access control models are introduced as conceptual security patterns. In addition, six access control architecture patterns are described. Three of these indicate how to control access to services: Single Access Point, Check Point, and Security Session. Two user interface patterns with respect to access control are also introduced: Full Access with Errors, and Limited Access. These patterns are based on proposals in [15] and [16].

Even if access control of data works well, attackers may read the data by wiretapping it through the network. In this case, the confidentiality property of the data is violated. We can avoid this by using Secure Channels [2] or Secure Pipe [18]. Jürjens proposed a secure data transfer pattern denoted by UMLsec [5].

Integrity of assets is achieved by refusing to permit modifications by unauthorized people. A digital signature is attached to prove the fact that an authorized person modifies it. In [19], the Signed Authenticated Call Pattern for Voice over IP Networks (VoIP) is introduced to guarantee the integrity of calls.

Firewall patterns can be applied to ensure the availability of services. The following three firewall patterns are introduced in [2] for each implementation level:

- IP Level: Packet Filter Firewall pattern
- Transport Level: Proxy Firewall pattern
- Service Level: Stateful Firewall pattern

Firewalls can not only mitigate attacks but also satisfy confidentiality of services with an access policy, in addition to providing for availability.

For accountability, there are five accounting patterns in [2]: Security Accounting Requirements, Audit Requirements, Audit Trail, Intrusion Detection Requirements, and Non-Repudiation Requirements. These are process patterns.

3.2.2 Bridge between security design patterns and security properties

Almost all of the patterns described in section 3.2.1 do not clearly indicate their relations with the security requirements, such as security properties and strategy.

Weiss proposed an approach to determining the relation between design patterns and security properties by using a non-functional requirements (NFR) model [20]. A link from a pattern to a security property indicates the contribution of the property denoted by a goal of NFR. Weiss modeled security properties, the reasons for a pattern, security design, and the relations among these. Therefore, we can apply reasoning to security properties. For example, if pattern A is used by pattern B, then the properties held by pattern A might be satisfied by pattern B.

3.2.3 Domain-specific design patterns

If the design patterns in [17] are implemented on a distributed object framework, such as CORBA or DCOM, network-based attacks might result. There have been several proposals for making design patterns secure. In [21], a Secure Broker pattern is introduced for a Broker pattern [22]. The security patterns book [2] introduces a Controlled Object Factory pattern in which secure objects are created with respect to an Abstract Factory Design pattern [17]. In addition, the Secure Service Facade pattern [18] is a secure extension of the Session Facade pattern [23].

There are security patterns for the OS level [2], [24], [25] as well. In [2], eight patterns are introduced for operating system access control. Hafiz proposed a secure pre-forking pattern by which a task can be forked securely and efficiently in a multitasking environment [25].

There is a security pattern catalogue for the Web and J2EE in [18]. In this book, 23 security design patterns are introduced for designing Web applications by using UML and Java. Fig. 2 illustrates the class diagram of the Secure Logger pattern with the Secure Log Store strategy.

Security patterns for privacy have also been proposed [26], [27]. In [26], a protection pattern using a client-side proxy to prevent unnecessary leakage of private information is shown. In addition, Romanosky et al. proposed three privacy patterns, such as “informed consent for web-based transactions,” in [27].

Application-specific security patterns have been also proposed. For instance, security patterns for VoIP are described in [19].

3.3 Security patterns for implementation phase

Implementation is the task of writing software that satisfies a given design. To construct secure software, programs must correctly implement various mechanisms that support security. Even if software has the correct security features in its design, the actual program could contain serious security bugs. Software consisting of vulnerable programs cannot be considered secure. To prevent implementation-level attacks, it is sufficient to write program code that does not include security bugs.

An implementation flaw (bug) is a mistake made while a programmer writes a software program. A security flaw could pose a potential security risk. Unfortunately, not all programmers have knowledge sufficient to create a secure system. Although some are experts on security, they are not perfect and sometimes make mistakes. Therefore, many researchers have developed guidelines that support the sharing of knowledge about how to write secure code or that help the detection of security flaws existing in code. We consider these traditional guidelines immature, however, since programmers do not easily or adequately reuse them. The guidelines should be more sophisticated. We need implementation-level security patterns, which are semi-structured documents or standard vocabularies that explicitly express these guidelines and contain artifacts associated with them. Such patterns enable programmers to be familiar with various kinds of secure coding techniques and to apply suitable techniques while writing software.

3.3.1 Secure programming guidelines

Many implementation flaws related to security have been documented in a variety of formats for years. For example [28], presents 18 implementation rules that programmers should note in order to eliminate common security programming problems. These rules are all informally described in natural language. Similar rules are shown in [29], which provides lists, including a total of more than 20 practices, of recommended practices (good practices) in six categories, and one list of 23 flawed practices (bad practices).

A set of design and implementation guidelines emphasizing the programmer’s viewpoint for writing secure programs is, in general, called secure programming [30]. provides well-known guidelines for writing secure programs for Linux and Unix system. This article describes not only guidelines for several security vulnerabilities (input validation, buffer overflow, etc.) but also language-specific issues for C/C++, Perl, Python, shell scripting (sh and csh derivatives), Ada, Java, Tcl, and PHP [31]. introduces 12 rules for writing security-critical Java code. Jslint [32] is an automated tool that can detect security vulnerabilities by using these 12 rules. In addition, [33] takes up six topics on security vulnerabilities often caused by Java pro-

grammers and presents security-specific coding guidelines to minimize the likelihood of such vulnerabilities.

Almost all collections of secure programming guidelines emphasize the topics of input validation and buffer overflow. Some of them touch on access control, cryptography, authentication and authorization, networking, and I/O (input/output). For example, [34] and [35] both provide guidelines specific to C and C++ [36]. targets Java and its platform, and [37] targets Microsoft .NET [38]. focuses on network applications.

These guidelines have been pragmatically collected from actual programming experiences. Each of them shows concrete security flaws, vulnerabilities and exploits that might be caused by the flaws, and mitigation methods to prevent or limit exploits against vulnerabilities. Unfortunately, there is no overriding, consistent method for documenting these guidelines, even though they contain descriptions that should be described through secure patterns.

3.3.2 Attack pattern catalog

To the best of our knowledge, there are two books [39], [40] that provide semi-structured documents related to implementation-level (plus design-level) security. Both books focus attention on how to break software. Thus, these seem to be collections of attack patterns [39]. [41] presents 19 attack patterns and examples for practically applying these attacks to actual applications (Microsoft Windows Media Player, Mozilla, and OpenOffice.org). [40] shows 24 attack patterns for web applications.

In these books, each attack pattern consists of a name (or concise sentence) and sections starting with a special keyword, which is either WHEN, WHAT, or HOW. A WHEN section explains a situation in which the attack might be successful. From the perspective of programmers (developers), this shows the properties of the developed program. The properties must be tested before the program is actually used. A WHAT section indicates causal faults, i.e., what faults make the attack successful. The programmers remove faults (probably implementation bugs or design flaws) described in this section. A HOW section is mainly used to explain how to determine whether security is compromised and how to conduct the attack. Although these attack patterns do not directly help coding, they are all useful for improving the implementation of developed programs.

3.3.3 Secure refactoring

As our focus shifts from the literature on security to software refactoring, we find that several refactorings are related to security characteristics. For example, Fowler's catalogue [42] contains several refactorings that can remove the security flaws of existing object-oriented programs. Refactoring is the process of altering the internal structure of existing code without changing its external (observable) behavior [42], [43]. A cataloged refactoring seems to be a kind of modification (transformation) pattern for a design specification or code, since its documentation is well-formed in describing recurrent problems (requirements) and their solutions (plus consequences).

In [42], the Encapsulate Field refactoring converts the accessibility setting of a field from public to private in order to make it harder for any client (an attacker in most cases) to access the value of the encapsulated field. If such a field is set at development time and will be never altered, the Remove Setting Method refactoring can be further applied, removing the setting method for the field and declaring the field final. A private final field without a setting method prevents a client from changing the value of the field. The Encapsulate Collection refactoring replaces the setting method for a collection (e.g., an array, list, or set) by adding or removing methods for the collection, and it rewrites the getting method of the collection so that it returns either an unmodifiable view or a copy. The code after this transformation does not allow any client to freely change the contents of the encapsulated collection. In addition, the Encapsulate Classes with Factory refactoring in Kerievsky's catalog [44] reduces the possibility of malicious access from any client by hiding classes that do not need to be publicly visible. No attacker can access sensitive data stored in the encapsulated classes, since attackers have no way to know the names of such classes or to obtain references to their instances.

One of us successfully developed four new refactorings increasing the security level of existing code, which are called secure refactorings [45]. These refactorings change the internal structure of existing software to make it more secure without changing its observable behavior. Each refactoring has a name, motivation, mechanics, and examples. The motivation describes why the refactoring should be applied and the result of its application. The mechanics demonstrate a situation in which the refactoring should be applied (called the *security smell*), and they present concise procedures for how to carry it out. The examples show code snippets before and after application of the refactoring.

In [45], the Introduce Immutable Field refactoring protects any field from malicious modification. The Replace Reference with Copy refactoring prevents attackers from changing internal data by exploiting an obtained reference. The Prohibit Overriding refactoring changes code so that it cannot allow an arbitrary (possibly malicious) class to redefine a method exist-

ing in the code. The Clear Sensitive Value Explicitly refactoring explicitly clears the value of a variable storing sensitive data at the earliest possible time in order to protect such information from theft. All the transformations of these secure refactorings can be codified into an automated tool.

3.4 Security pattern engineering

From the pattern lifecycle point of view, activities using software patterns can be classified into two processes: extraction and application.

- The extraction process consists of several activities: (E_1) finding a pair of a recurring problem and its corresponding solution from knowledge and/or experiences of software development; (E_2) writing the found pair with forces in a specific pattern format (such as that shown in section 2.2); (E_3) reviewing and revising the written pattern; and (E_4) publishing the revised pattern via some public (or private, if necessary) resource such as a book, the WWW, or specific repositories.
- The application process also consists of several activities: (A_1) recognizing contexts and problems in software developments; (A_2) selecting software patterns, from public or private resources, that are thought to be useful for solving the recognized problems; (A_3) applying the selected patterns to the target problem; and (A_4) evaluating the application result.

These activities are thought to be common for security patterns. Therefore in the following, we show several engineering achievements and ongoing research on utilizing security patterns by relating those achievements and research to the above pattern activities.

3.4.1 Representation of security requirements and patterns

As shown in section 2.2, software patterns are loosely structured documents. Most security patterns are structured by using the GoF [17] or POSA [22] formats that have been used for describing design or architecture patterns. This is somewhat reasonable because many security patterns provide processes for creating particular design structures. These general formats, however, are not specific enough to capture security requirements as target problems. The nature of security patterns regarding their formats makes it difficult to select appropriate security patterns satisfying the target security requirements.

To overcome this situation, research [46], [47] has focused on modeling and describing security patterns formally and precisely in order to develop a tool for supporting a mapping mechanism from security patterns to requirements and for retrieving patterns based on the map.

Schumacher proposed an approach for implementing a security pattern search engine [46]. Using a symbolic ontology representation language F-Logic [48], this approach defines a knowledge base composed of a security ontology, mappings between security concepts and security pattern elements, and inference rules among patterns (such as pattern p_1 requires another pattern p_2 in its solution). By using the knowledge base, developers can retrieve security patterns that meet given queries, such as contexts and problems. Moreover, this approach provides a way to find all required security patterns according to a pattern hierarchy, and to detect conflicting and alternative patterns.

In addition, Supaporn et al. proposed an approach to construct extended-BNF-based grammars of security requirements from security patterns in order to translate from the security needs of any project or organization to system security requirements [47]. Moreover, they also proposed a prototyping tool for defining security requirements based on the constructed grammars. This tool supports developers in selecting types of security requirements by providing lists of security properties and related security patterns, and in instantiating the selected security patterns by displaying forms for the developers to fill in the necessary information.

The research is helpful mainly for supporting activities E_2 and A_2 in the above-mentioned classification.

3.4.2 Classification of security patterns

The large number of security patterns makes selection of the right pattern for a job a daunting task [49]; we need guidance for developers on how to select appropriate patterns. To provide good guidance for security pattern selection, it is necessary to clarify the relations among patterns. There has been some such research [2], [50]–[52] on classifying security patterns.

Konrad et al. proposed a classification method of organizing security patterns by using the aspect types of the patterns (creational, structural, or behavioral, as defined in [17]) and the abstraction level (network, host, or application) [50]. Similar classification can be found elsewhere [2], [51]. Rosado et al. related typical security requirements to security patterns, and classified patterns into architectural and design patterns [51].

Hafiz et al. proposed several ways to classify and organize security patterns [52], such as a classification based on the CIA model [53], which describes three key aspects of security (confidentiality, integrity, and availability), a tabular classification based on patterns' application contexts and the Zachman framework [54], and a classification based on the STRIDE model [55],

which categorizes different types of threats.

All of the above-mentioned classifications are conducted manually through experts consideration. In contrast, there is a challenge [56] to classify security patterns automatically by applying a relation extraction technique [57]. There is a tradeoff between scalability and validity. Manual classification techniques are thought superior in terms of validity. In contrast, automatic classification techniques are superior in terms of scalability.

These classifications are helpful mainly for supporting activity A_2 above.

3.4.3 Repository of security patterns

In relation to the above section, there are several repositories and inventories collecting and categorizing a number of software patterns including security patterns. Some of these use existing classification schemes, such as those shown in section 3.4.2, and are helpful for supporting activity A_2.

Yskout et al. developed an inventory in which each security pattern is classified as an architectural pattern or a design pattern [58]. The relationships among patterns and the quality tradeoff for each pattern were analyzed and explicitly described.

Moreover, there are several online repositories of all kinds of software patterns, such as patternshare [59] and the Portland Pattern Repository [60]. These online repositories are useful for supporting activities E_4 and A_2; however, these are not specific to security patterns, so mechanisms for security-specific classification and search are unavailable.

3.4.4 Quality analysis of security patterns

There is also research on assessing the overall quality of security patterns, which mainly supports activities E_3 and A_2.

Halkindis et al. qualitatively analyzed a set of security patterns by using Viega and McGraw's ten security principles [61] as evaluation criteria [62]. Similarly, Konrad et al. applied the same evaluation criteria to another set of security patterns [50].

Heyman et al. used different quantitative criteria for a larger set of security patterns [49]. They assessed security patterns according to three dimensions: appropriateness (i.e., whether a pattern can be used constructively for developing the architecture or designing an application), the quality of the pattern's documentation, and the distribution of security patterns over the security objectives. For example, they found that the most well-represented objective of existing security patterns is access control [49].

3.4.5 Development methodology with security patterns

As mentioned previously in this paper, there are a number of security patterns and catalogs that organize related security patterns; most of them, however, do not provide any clear information about when to apply them within the entire software/system life cycle process. To complement such information, a development methodology utilizing security patterns is needed.

Fernandez et al. proposed a methodology that incorporates security patterns in all the life cycle stages [63], [64]. This methodology proposes guidelines for incorporating security from the requirements stage through analysis, design, implementation, testing, and deployment [64].

There are other methodologies for developing secure software systems by using security patterns together with other tools or techniques [63]. For example, these include aspect-oriented security design methodologies, in which the security aspects are modeled using patterns and weaved into functional models [65], [66]; a security engineering process based on problem frames [9] with security patterns [14]; and an approach using security patterns and UMLSec [67] (a UML extension for modeling security constraints) in the secure software development life process [68].

These methodologies directly support the entire process of security pattern application (A_1 through A_4) in our above classification.

4 Discussions

In this section, we first discuss the ease of pattern use, and the effectiveness and the sufficiency of security patterns for security engineering; then, we show what kinds of security patterns will be needed in the future. We then discuss how previous security patterns can be adapted to system models that point to future research directions.

4.1 Ease of security pattern use

There are interrelations between the quality and the ease of use of patterns. As described in section 3.4.4, Heyman et al. defined the appropriateness and the quality of the pattern's documentation [49]. For appropriateness, they used two kinds of definitions: a negative definition (patterns vs. concepts and mechanisms), and a positive definition (scoped problem with a time-proven, constructive solution). Consequently, they found that 55% of patterns are classified as core patterns, 35% are guidelines and principles, and 10% are process activities. In addition, for the quality of a pattern's documentation, they defined a good description of the problem and forces, the solution, and the consequences and assigned scores to these in order to measure the quality of patterns.

According to this classification, analysis process patterns are regarded as process activities rather than core patterns. For instance, the asset valuation pattern [2] does not include any solutions for security measures, although it illustrates a kind of domain analysis. As mentioned in section 3.1, however, we should define security requirements in the requirements phase. Therefore, it is useful to define process activities for the specification of typical security requirements and examples as security patterns.

The problem of analysis process patterns, described in section 3.1.1, does not indicate the specific context but a general one. Therefore, the specification is abstract, so that the application of patterns depends on engineer's experience rather than knowledge of patterns. This implies that it is difficult to apply such analysis process patterns to each problem. We need to make the patterns specific to particular domains.

In [49], one of the qualitative criteria for a pattern's documentation is the use of images such as UML diagrams. Patterns in the design phase are easily understood through such UML diagrams. For example, the solutions of the patterns in [18] are illustrated using class diagrams, such as that shown in Fig. 2, and sequence diagrams. The ease of understanding of patterns, however, does not imply ease of application of the patterns.

For example, it is easy to understand the secure pipe pattern [18], but its actual application is not easy. The problem is described as follows:

> *"You need to provide privacy and prevent eavesdropping and tampering of client transactions caused by man-in-the middle attacks."*

In the pattern, however, the definition of the private data and insecure networks that should be protected are not specified. As a consequence, instantiation of the pattern is not so easy. For ease of pattern application, analysis process patterns and methodologies for developing secure systems might be important.

Patterns in the implementation phase should be sophisticated, as described in section 3.3. Many patterns in this phase only indicate techniques and guidelines in term of the solution, while the problem and context are not clear. In the problem and context sections of patterns, the vulnerabilities should be described concretely.

4.2 Effectiveness of security patterns

Security patterns are intended to be secure and to use security functions efficiently. For example, the authentication enforcer pattern in [18] aims to use authentication functions efficiently with requests from multiple clients. This pattern effects the reusability and maintainability of authentication functions. The effectiveness can be measured the usual metrics. For example, Vokac explored the relation between defect frequency and the application of five design patterns [69].

For the effectiveness of security, we need to define a security metric, such as the frequency of security violations or the inconsistency of security requirements and a function. After applications of security patterns, threats might be mitigated, and a system might maintain a proper security level.

4.3 Sufficiency of security patterns

The table in Fig. 3 illustrates the relative density of security patterns proposed for each phase in software development. In this table, each column and row specifies a security concept and a phase, respectively. Each cell in the table shows an activity with respect to a security concept and how many security patterns appear. A "+" indicates that many security patterns exist, with "++" meaning a relatively larger number.

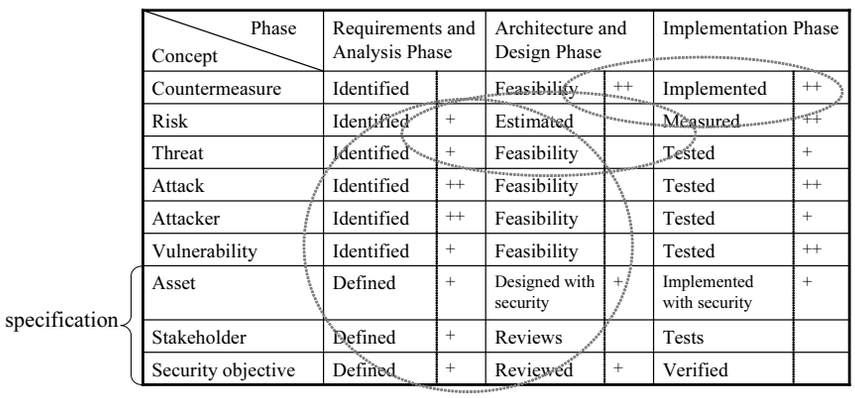
We can infer the following facts from the data in the table:

- In the requirements and analysis phases, there are many attack patterns. In contrast, there are few patterns for countermeasures.
- In the architecture and design phases, there are relatively small numbers of attack patterns and patterns for security specification.
- In the implementation phase, there are many attack patterns and patterns for countermeasures but only a small number of patterns for security specification.

These results seem to imply that we need more patterns for the areas with smaller numbers of patterns in the table in order to improve sufficiency. We do not know, however, whether all kinds of patterns are useful. In [46], Schumacher shows possible overlaps where we need to clarify the security concepts for each phase. The bubbles in Fig. 3 indicate the overlaps [3] which illustrate areas important to other phases. In other words, we can bridge the gap from security requirements to implementation of security functions seamlessly by using patterns in the overlapping areas. In addition, pattern languages might play an important role in connecting phases with each other.

Consequently, we realize that we need patterns for risk and attacks in the design phase to bridge the gap between the requirement phase and the design phase. Additionally, these patterns are needed to bridge the

[3] We omit some phases and concepts because of space limitations.



Concept \ Phase	Requirements and Analysis Phase		Architecture and Design Phase		Implementation Phase	
Countermeasure	Identified		Feasibility	++	Implemented	++
Risk	Identified	+	Estimated		Measured	++
Threat	Identified	+	Feasibility		Tested	+
Attack	Identified	++	Feasibility		Tested	++
Attacker	Identified	++	Feasibility		Tested	+
Vulnerability	Identified	+	Feasibility		Tested	++
Asset	Defined	+	Designed with security	+	Implemented with security	+
Stakeholder	Defined	+	Reviews		Tests	
Security objective	Defined	+	Reviewed	+	Verified	

specification

Fig. 3 Security concepts and security patterns in development phases.

gap between the security requirements and implementation.

Nevertheless, it seems difficult to find threats, attacks, and vulnerabilities of systems at the design level. The reason is that it is difficult to find the influence of attacks at the design level when the the inner details of a system have not yet been implemented. Therefore, we need to establish a methodology for designing attack patterns at the design level. We might consider a method using the monitoring of attackers in a maintenance phase, or designing attacks in a manner similar to system design.

4.4 Unifying security patterns and system models

In [1], Devanbu showed the challenges for the next generation of software engineering for security. The following are two of the challenges:

- Unifying security with system engineering
- Unifying security and system models

The enrichment of security patterns seems to meet the former challenge. We need a deeper analysis mechanism or formalism of systems for security patterns, however, in order to support the analysis and selection of security features.

For the latter challenge, we need compatibility with system models. The following models have been proposed for security modeling in early phases.

- Problem-Frame-based models: abuse frame [70], anti-requirements [71], malicious subject, etc
- Tropos-based models: i*/Tropos, Secure Tropos [72], attacker agents, trust relations, etc
- UML-based models: abuse case, misuse Case [73], misuse actions [74], etc

In the future, we might need to integrate the above models into security patterns.

UML-based models are widely used for design. For modeling security concerns, models such as UMLsec [67] and SecureUML [75] have been proposed, so we need to adapt such models to security patterns.

There is existing research in which UML-based models are adapted to security patterns [5], [50], [76]. In [5], [50], security properties can be checked precisely by using formal methods. In [76], we can find the applicable parts of security patterns, because the contexts of patterns are denoted using a deployment diagram and a sequence diagram without security functions. This means that systems before security functions are attached will be matched with the diagrams in context.

5 Conclusions

In this paper, we have categorized security patterns according to software development phases and then analyzed the patterns with respect to security concepts and discussed their sufficiency. Consequently, we have showed the potential needs for security patterns with attack design. In addition, we gave a research direction for modeling in security patterns.

Our future work will include a proposal of a design method and a model for security patterns with attacks.

References

[1] P. T. Devanbu and S. Stubblebine, “Software engineering for security: a roadmap,” in *Proceedings of the Conference on The Future of Software Engineering*, pp.227–239, 2000.

[2] M. Schumacher, E. B. Fernandez, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating Security And Systems Engineering*, John Wiley & Sons Inc, 2006.

[3] E. Houg, N. R. Mead and T. R. Stehney, “Security quality requirements engineering (square) methodology,”

Technical Report CMU/SEI-2005-TR-009, CMU/SEI, 2005.

[4] P. Bresciani, P. Giorgini, F. Giunchiglia, and J. Mylopoulos, "Tropos: An agent-oriented software development methodology," *JAAMAS*, vol.8, no.3, pp.203–236, 2004.

[5] J. Jürjens, G. Popp, and G. Wimmel, "Towards using security patterns in model-based system development," in *Proceedings of PLoP 2002 Conference*, 2002.

[6] C. Alexander, *The Timeless Way of Building*, Oxford University Press, 1979.

[7] F. Buschmann, K. Henney, and D.C. Schmidt, *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*, John Wiley & Sons Inc, 2007.

[8] M. Kis, "Information security antipatterns in software requirements engineering," in *the PLoP 2002 conference*, 2002.

[9] M. A. Jackson, *Problem Frames: Analysing and structuring software development problems*, Addison Wesley, 2000.

[10] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone, "Modelling security requirements through ownership, permission and delegation," in *13th IEEE International Conference on Requirements Engineering 2005*, pp.167–176, 2005.

[11] H. Mouratidis, M. Weiss, and P. Giorgini, "Modelling secure systems using an agent-oriented approach and security patterns," *International Journal of Software Engineering and Knowledge Engineering*, vol.16, no.3, pp.471–498, 2006.

[12] D. Hatebur, M. Heisel, and H. Schmidt, "Security engineering using problem frames," in *Proceedings of the International Conference on Emerging Trends in Information and Communication Security (ETRICS)*, vol.3995, pp.238–253, Springer Berlin, Heidelberg, 2006.

[13] D. Hatebur, M. Heisel, and H. Schmidt, "A pattern system for security requirements engineering," in *Proceedings of the International Conference on Availability, Reliability and Security* (*AReS*), pp.356–365, IEEE, 2007.

[14] D. Hatebur, M. Heisel, and H. Schmidt, "A security engineering process based on patterns," in *Proceedings of the International Workshop on Database and Expert Systems Applications (DEXA)*, pp.734–738, IEEE, 2007.

[15] J. Yoder and J. Barcalow, "Architectural patterns for enabling application security," in *Proceedings of PLoP '97 Conference*, 1997.

[16] E. B. Fernandez and R. Pan, "A pattern language for security models," in *Proceedings of PLoP 2001 Conference*, 2001.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995.

[18] C. Steel, R. Nagappan, and R. Lai, *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*, Prentice Hall, 2005.

[19] E. B. Fernandez, J. C. Pelae, and M. M. Larrondo-Petrie, "Security patterns for voice over ip networks," in *Proceedings of the International Multi-Conference on Computing in the Global Information Technology (ICCGI'07)*, 2007.

[20] M. Weiss, *Integrating Security and Software Engineering: Advances and Future Vision*, Chapter VI: Modelling Security Patterns Using NFR Analysis, pp.127–141, Idea Group Publishing, 2006.

[21] P. Morrison and E. B. Fernandez, "Securing the broker pattern," In *Proceedings of the 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006)*, 2006.

[22] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, Wiley, 1996.

[23] D. Alur, J. Crupi, and D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies, Second Edition*, Prentice Hall, 2003.

[24] E. B. Fernandez, T. Sorgente, and M. M. Larrondo-Petrie. "Even more patterns for secure operating systems," in *Proceedings of PLoP 2006 Conference*, 2006.

[25] M. Hafiz, "Secure pre-forking - a pattern for performance and security," in *Proceedings of PLoP 2005 Conference*, 2005.

[26] M. Sadicoff, M. M. Larrondo-Petrie, and E. B. Fernandez, "Privacy-aware network client pattern," in *Proceedings of PLoP 2005 Conference*, 2005.

[27] S. Romanosky, A. Acquisti, J. Hong, L. F. Cranor, and B. Friedman, "Privacy patterns for online interactions," in *Proceedings of PLoP 2006 Conference*, 2006.

[28] M. Bishop, *Computer Security: Art and Science*, Chapter 29: Program Security, pp.869–921, Addison Wesley, 2003.

[29] M. G. Graff and K. R. Wyk, *Secure Coding: Principles and Practices*, Chapter 4: Implementation, pp.99–123, O'Reilly, 2003.

[30] D. A. Wheeler, "Secure Programming for Linux and Unix HOWTO," 1999. http://www.dwheeler.com/secure-programs/.

[31] G. McGraw and E. Felten, "Twelve rules for developing more secure java code," 1998; http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html.

[32] J. Viega, G. McGraw, T. Mutdosch, and E.W. Felten, "Statically scanning java code: Finding security vulnerabilities," *IEEE Software*, vol.17, no.5, pp.68–74, 2000.

[33] Sun Microsystems, Security code guidelines, 2000. http://java.sun.com/security/seccodeguide.html.

[34] J. Viega and M. Messier, *Secure Programming Cookbook for C and C++*, O'Reilly, 2003.

[35] R. C. Seacord, *Secure Coding in C and C++*, Addison Wesley, 2006.

[36] S. Oaks, *Java Security, 2nd ed.* Addison-Wesley, 2001.

[37] M. Howard and D. LeBlanc, *Writing Secure Code, Second Edition*, Microsoft Press, 2002.

[38] C. Wysopal, L. Nelson, D. D. Zovi, and E. Dustin, *The Art of Software Security Testing*, Addison-Wesley, 2006.

[39] J. A. Whittaker and H.H. Thompson, *How to Break Software Security*, Addison Wesley, 2001.

[40] M. Andrews and J. A. Whittaker, *How to Break Web Software*, Addison-Wesley, 2006.

[41] G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code*, Addison-Wesley, 2004.

[42] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[43] W. F. Opdyke, *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, 1992.

[44] J. Kerievsky, *Refactoring to Patterns*, Addison-Wesley, 2004.

[45] K. Maruyama, "Secure refactoring: Improving the security level of existing code," in *Proc. Int'l Conf. Software and Data Technologies (ICSOFT 2007)*, pp.222–229, 2007.

[46] M. Schumacher, *Security Engineering With Patterns: Origins, Theoretical Models, and New Applications*, Springer, 2003.

[47] K. Supaporn, N. Prompoon, and T. Rojkangsadan, "An approach: Constructing the grammar from security pattern," in *Proc. 4th International Joint Conference on Computer Science and Software Engineering (JCSSE2007)*, 2007.

[48] M. Kifer, G. Lausen, and J. Wu, "Logical foundations of object oriented and frame based languages," *Journal of ACM*, vol.42, pp.741–843, 1995.

[49] T. Heyman, K. Yskout, R. Scandariato, and W. Joosen, "An analysis of the security patterns landscape," in *3rd International Workshop on Software Engineering for Secure Systems (SESS07), Proc. 29th International Conference on Software Engineering Workshops (ICSEW'07)*, IEEE CS, 2007.

[50] S. Konrad, B. H. C. Cheng, L. A. Campbell, and R. Wassermann, "Using security patterns to model and analyze security requirements," in *International Workshop on Requirements for High Assurance Systems*, 2003.

[51] D. G. Rosado, C. Gutierrez, Eduardo Fernandez-Medina, and M. Piattini, "Security patterns related to security requirements," in *Proc. 4th International Workshop on Security in Information Systems (WOSIS)*, 2006.

[52] M. Hafiz, P. Adamczyk, and R.E. Johnson, "Organizing security patterns," *IEEE Software*, vol.24, no.4, pp.52–60, 2007.

[53] Commission of European Communities, Information technology security evaluation criteria, version 1.2, 1991.

[54] J. A. Zachman, "A framework for information systems architecture," *IBM Systems Journal*, vol.26, no.3, 1987.

[55] F. Swiderski and W. Snyder, *Threat modeling*, Microsoft Press, 2004.

[56] A. Kubo, H. Washizaki, and Y. Fukazawa, "Extracting relations among security patterns," *Submitted to 1st International Workshop on Software Patterns and Quality (SPAQu'07)*.

[57] A. Kubo, H. Washizaki, A. Takasu, and Y. Fukazawa, "Extracting relations among embedded software design patterns," *Journal of Design & Process Science*, vol.9, no.3, pp.39–52, 2005.

[58] K. Yskout, T. Heyman, R. Scandariato, and W. Joosen, "An inventory of security patterns," in *Technical Report CW-469*, Katholieke Universiteit Leuven, Department of Computer Science, 2006.

[59] Microsoft, Patternshare; http://patternshare.org/.

[60] Cunningham & Cunningham, Inc., Portland pattern repository; http://c2.com/ppr/.

[61] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, 2001.

[62] S. T. Halkidis, A. Chatzigeorgiou, and G. Stephanides, "A qualitative evaluation of security patterns," in *Proc. International Conference on Information and Communications Security (ICICS)*, 2004.

[63] E. B. Fernandez, M.M. Larrondo-Petrie, T, Sorgente, and M. VanHilst. *Integrating Security and Software Engineering: Advances and Future Vision*, Chapter V: A methodology to develop secure systems using patterns, pages 107–126, Idea Group Publishing, 2006.

[64] E. B. Fernandez, "Security patterns," in *Procs. Eigth International Symposium on System and Information Security (SSI'2006), Keynote talk*, 2006.

[65] G. Georg, I. Ray, and R. France, "Using aspects to design a secure system," in *Proc. Eighth IEEE International Conference on Engineering of Complex Computer Systems*, 2002.

[66] I. Ray, R. France, N. Li, and G. Georg, "An aspect-based approach to modeling access control concerns," *Information and Software Technology*, vol.46, no.9, pp.575–587, 2004.

[67] J. Jürjens, *Secure Systems Development with UML*, Springer, 2004.

[68] A. Apvrille and M. Pourzandi, "Secure software development by example," *IEEE Security & Privacy*, vol.3, no.4, pp.10–17, 2005.

[69] M. Vokac, "Defect frequency and design patterns: an empirical study of industrial code," *Transactions on Software Engineering*, vol.30, no.12, pp.904–917, 2004.

[70] L. Lin, B. Nuseibeh, D. Ince, M. Jackson, and J. Moffett, "Analysing security threats and vulnerabilities using abuse frames," *Open University Technical Report No:2003/10*, 2003. Abuse Frame.

[71] R. Crook, D. Ince, L. Lin, and B. Nuseibeh, "Security requirements engineering: When anti-requirements hit the fan," in *Proceeding of the 10th Requirements Engineering Conference (RE'02)*, pp.9–13, 2002.

[72] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone, "Requirements engineering meets trust management: Model, methodology, and reasoning," in *Proc. of iTrust'04*, LNCS 2995, pp.176–190, Springer-Verlag, 2004.

[73] P. Hope, G. McGraw, and A.I. Anton, "Misuse and abuse cases: Getting past the positive," *IEEE Security & Privacy*, vol.2, no.3, pp.90–92, 2004.

[74] E. B. Fernandez, M. VanHilst, M.M. Larrondo, and S. Huang, "Defining security requirements through misuse actions," in *IFIP International Federation for Information Processing*, pp.123–137, 2006.

[75] T. Lodderstedt, D. A. Basin, and J. Doser, "SecureUML: A UML-based modeling language for model-driven security," in *Proceedings of the 5th International Conference on The Unified Modeling Language*, pp.426–441, 2002.

[76] N. Yoshioka, S. Honiden, and A. Finkelstein, "Security patterns: a method for constructing secure and efficient inter-company coordination systems," in *Proceedings of Enterprise Distributed Object Computing Conference 2004 (EDOC'04)*, pp.84–97, 2004.

Nobukazu YOSHIOKA

Nobukazu YOSHIOKA is a researcher at the National Institute of Informatics, Japan. Dr. Nobukazu Yoshioka received his B.E degree in Electronic and Information Engineering from Toyama University in 1993. He received his M.E. and Ph.D. degrees in School of Information Science from Japan Advanced Institute of Science and Technology in 1995 and 1998, respectively. From 1998 to 2002, he was with Toshiba Corporation, Japan. From 2002 to 2004 he was a researcher, and since August 2004, he has been an associate professor, in National Institute of Informatics, Japan. His research interests include agent technology, object-oriented methodology, software engineering for secure systems, and software evolution. He is a member of the Information Processing Society of Japan (IPSJ), and Japan Society for Software Science and Technology.

Hironori WASHIZAKI

Hironori WASHIZAKI is an assistant professor at National Institute of Informatics, Tokyo, Japan. He is also an assistant professor at the The Graduate University for Advanced Studies, Japan. He obtained his Doctor's degree in Information and Computer Science from Waseda University in 2003. His research interests include software reuse, quality assurance and patterns. He has published more than 35 research papers in refereed international journals and conferences. He received Takahashi Encouraging Award from JSSST in 2004, SES2006 Best Paper Award from IPSJ/SIGSE, and Yamashita Memorial Research Award from IPSJ in 2007. He has served as member of program committee for several international conferences and workshops including REP'04, ASE'06, Profes'04-08, APSEC'07, SPACE'07, JCKBSE'08 and ISA'08, as workshop co-chair for ASE'06, as publicity chair for APSEC'07, and as program co-chair for SPAQu'07. He is a member of IEEE, ACM, IEICE, JSSST, IPSJ, DBSJ, JUSE, JSQC and Hillside Group.

Katsuhisa MARUYAMA

Katsuhisa MARUYAMA received B.E. and M.E. degrees in electrical engineering and a ph.D in information and computer science from Waseda University, Japan, in 1991, 1993, and 1999, respectively. He is a professor of the Department of Computer Science at College of Information and Engineering, Ritsumeikan University. He has worked for NTT (Nippon Telegraph and Telephone Corporation) and NTT Communications Corporation before he joined Ritsumeikan University. He was a visiting researcher at Institute for Software Research (ISR) of University of California, Irvine (UCI) from 2003 through 2004. His research interests include software refactoring, program analysis, software reuse, object-oriented design and programming, and software development environments. He is a member of the IEEE Computer Society, ACM, IPSJ, IEICE, and JSSST.