Requirements-based Access Control Analysis and Policy Specification (ReCAPS)

Qingfeng He and Annie I. Antón
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8207 USA
{qhe2, aianton}@eos.ncsu.edu

Abstract. Access control is a mechanism for achieving confidentiality and integrity in software systems. Access control policies (ACPs) are security requirements that define how access is managed and the high-level rules of who, under what conditions, can access what information. Traditionally, access control policies are often specified after a system is designed and deployed. Because ACP specification is typically isolated from requirements analysis, it may result in policies that are not in compliance with system requirements. We present a Requirements-based Access Control Analysis and Policy Specification (ReCAPS) approach for deriving ACPs from various information sources. It provides prescriptive guidance for how to specify both function-level and database-level ACPs. This approach helps clarify ambiguities, resolve conflicts, and it provides traceability support, ensuring consistency among software artifacts. We employ examples from three projects to present the ACP specification process and associated heuristics.

1. Introduction

The function of access control (AC) is to ensure every access to a system and its resources are controlled according to a set of predefined policies [42]. It is one of the major security mechanisms used to achieve confidentiality, integrity and data privacy in software systems [17]. We use these terms as follows: Confidentiality means that information is not disclosed to unauthorized persons, processes or devices. Integrity means that unauthorized persons, processes or devices cannot modify information. Privacy means that data is protected so that it is used only for authorized business purposes, based on legal requirements, corporate policies and end-user choices.

Two major challenges in access control analysis are: (a) defining policies to control users’ access to the system and its resources, and (b) ensuring the resulting policies comply with the system requirements. Methodological support is needed to guide software engineers as they specify a system’s access control policies (ACPs). Access control analysis entails analyzing system requirements, organizational security and privacy policies, and organizational structures to specify access control policies. Defining and deploying ACPs is both a conceptually and practically complex process because software systems can have many users performing various tasks and many resources that need to be protected [40, 41]. Organizational complexity presents another challenge — it is difficult to identify and agree upon a common set of roles and associated permissions within an organization that may have hundreds of roles to be considered.

Moffett et al. define security requirements as constraints on functional requirements [30]. Ideally, security requirements are analyzed and specified before system design rather than as an afterthought. These security requirements should drive ACP specification activities, but policy specification efforts often occur after systems are designed and deployed [8]. Because policy specification efforts are often isolated from requirements analysis, the resulting ACPs and requirements may not be compliant with one another. Our goal is to develop techniques that better align policies and system requirements.

Researchers recognize the need to bridge the gap between requirements engineering (RE) and complex ACP specification [8, 19, 22]. Existing RE approaches (e.g., KAOS [13], i* [43] and the analytical role modeling framework [8]) provide limited support as we discuss herein. We present an approach for deriving ACPs from various sources, including software requirements specifications (SRS), design documents, database designs, as well as high-level security and privacy policies. Our approach provides prescriptive guidance for specifying both function-level and database-level ACPs, improving the quality of the SRS and the software design.

Field studies have shown the importance of maintaining traceability [21]. However, it is challenging to maintain traceability in the face of the inevitable evolution that occurs during requirements and policy management. The approach presented herein features traceability support as a prominent design principle.

The rest of the paper is structured as follows. Section 2 summarizes relevant work. Section 3 presents our approach for deriving ACPs. Section 4 discusses three case studies. Section 5 details the ACP specification process and associated heuristics, using concrete examples from our case studies. Finally, Section 6 provides a discussion and summarizes our plans for future work.
2. Background and related work

2.1. Access control policies in security

An access control system is typically described in three ways: access control policies, access control models and access control mechanisms [42]. Access control policies are security requirements that describe how access is managed, what information can be accessed by whom, and under what conditions that information can be accessed [17]. These policies are enforced via a mechanism that mediates access requests and makes grant/deny decisions. Access control models provide a formal representation of an access control system. They provide ways to reason about the policies they support and prove the access control system’s security properties. Access control mechanisms define the low-level functions that implement the controls imposed by the policies. It must work as a reference monitor [42], a trusted component intercepting every request to the system.

ACPs can be broadly grouped into three main policy categories: Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Role-Based Access Control (RBAC). DAC policies enforce access control based on the identity of the requestor and the explicit rules specifying who can or cannot perform specific actions on specific objects. Early discretionary access control models, such as the access control matrix model [28, 20] and the HRU model [23], provide a basic framework for describing DAC policies. It is the users’ discretion to pass their privileges on to other users. Thus, DAC policies are vulnerable to Trojan Horse attacks [42].

MAC policies enforce access control based on the security classifications of subjects and objects. The lattice-based multilevel security policy [12], policies represented by the Bell-LaPadula model [6, 7] and the Biba model [4] are MAC policies. MAC policies protect indirect information leakages (e.g., Trojan Horse attacks), but are still vulnerable to covert channel attacks [42, 35]. RBAC policies employ roles to simplify authorization management for enforcing enterprise-specific security policies [17, 39]. The RBAC model is an alternative to traditional DAC and MAC models and has received increased attention in commercial applications, such as the Oracle 9i DBMS [10]. RBAC is now an American National Standard1.

Instead of considering ACP specification from a holistic, real-systems perspective as we advocate, current ACP specification research has a much narrower focus (e.g., uniform or flexible ways to specify ACPs [27], specifying ACPs for XML documents [18], etc.). There are few reported methods and experiences relating ACP specification in real software systems. In the RBAC literature2, researchers are investigating role engineering, the process of defining roles and privileges as well as assigning privileges to roles [9]. Several role-engineering approaches employ RE concepts. Fernandez and Hawkins suggest deriving the needed rights for roles from use cases [15]. Neumann and Strembeck propose a scenario-driven approach for engineering functional roles in RBAC [33]. Role engineering [9] is specific to RBAC, whereas our method is a more general approach for specifying ACPs.

2.2. Elements of access control policies

An access control policy is comprised of a set of access control rules. A rule can have various modes (e.g., permit/deny/obligate/refrain). This paper focuses on allow and deny rules. Allow rules authorize a subject to access a particular object. Deny rules explicitly prohibit a subject from accessing a particular object. When a subject requests to perform an action on an object, the corresponding rules are evaluated by an enforcement engine. A typical access control rule is expressed as a 3-tuple <subject, object, action>, such that a subject can perform some action on an object. A subject is an entity, such as a user or program agent that may access objects. An object is an entity, such as a data field, table, procedure or application to which access is restricted. An action is a simple operation (e.g. read or write) or an abstract operation (e.g. deposit or withdraw). We extend the typical AC rule 3-tuple with conditions and obligations as we now discuss.

An ACP may express additional conditions that must be satisfied before an access request can be granted. For example, in the healthcare domain if an access request is from the emergency room, then the request may be granted [3]. In this case, we can specify the location of the request is emergency room as a condition for the AC rule. Additionally, in the context of privacy protection, we may specify additional conditions to restrict access to personal data. For example, purpose is a standard entity in most privacy policies [34]. When a subject (e.g., a nurse) requests to perform an action on an object (e.g., medical records), the operation’s purpose should be bound to the purposes consented to by the data subjects. This is the purpose binding principle [16], which can be enforced by specifying ACP conditions as discussed in Section 5.

Obligations [5] are actions that must be fulfilled if a request to access an object is granted. Consider the requirement: require affiliates to destroy customer data after service is completed. Here, “destroy customer data” is an obligation that must be satisfied by affiliates. Obligation-based security policies can be enforced if they can be completely resolved within an atomic execution [37]. If the obligation is not an immediate action (e.g., a task to be executed in the future), monitoring and auditing its execution might be sufficient for enforcement [5].

In requirements specification, we are concerned with the actions for which each actor (subject) is responsible, the conditions under which each action can occur (constraints and pre-conditions) and the post-conditions (obligations) that must be satisfied. Each access control element can be mapped to a requirements specification element. This mapping suggests it is possible to derive ACPs from requirements to ensure they comply with each other.

2.3 Access control analysis in RE

RE researchers are investigating methods and tools to analyze and specify access-related security requirements.

Fontaine [19] employs KAOS, a goal-based requirements acquisition and elaboration method [13], to refine security requirements into specific authorization rules and ACPs expressed in Ponder — a language for specifying management and security policies for distributed systems [11]. Fontaine’s work is an important step towards requirements-level access control analysis for security policy specification. However, the method for mapping KAOS specifications to Ponder policies is not general in that not all types of Ponder policies can be generated from KAOS specifications. Fontaine has thus far only shown how to specify authorization and obligation policies [19], not refrain and delegation policies. ACPs not only come from requirements, but also security and privacy policies. Fontaine’s approach cannot specify ACPs from security and privacy policies whereas our approach can. Additionally, Fontaine’s approach does not ensure compliance between ACPs, requirements and design; in our approach, compliance is an important design principle (Section 3.2).

Liu et al. applied the i* framework [43], a goal-based requirements analysis method, to support access control analysis by modeling the dependencies among actors, tasks and a system’s resources [29]. However this approach is limited in that it assumes the roles and privileges have been previously derived. It provides no guidance as to how roles and privileges are identified, from where they originate, or how privileges are assigned to these roles. Moreover, it is difficult to model context and constraint information in the i* framework. These topics remain major challenges in access control analysis during RE.

Crook et al. proposed an analytical role-modeling framework to model ACPs [8]. This approach offers two contributions. First, the framework clarifies the need to model ACPs during requirements analysis. Second, the rationale for deriving roles based on organizational structures is very useful. Job positions in an organization can be mapped to roles in RBAC. Organizational and seniority hierarchies can be mapped to RBAC role hierarchies. This is common in identity management products in which individuals in a particular department and/or division are classified into a specific role and that role grants them the access rights to specific (e.g., bank) accounts. Deriving roles from organizational structures facilitates the user assignment and authorization management processes in access control.

Moffett et al. define security requirements to be constraints on functional requirements that are needed to achieve security goals [30]. They propose a set of core security requirements artefacts, in which goals, requirements and architecture form a hierarchical diagram. Moffett et al. also discuss the relationship between ACPs and RE, but do not provide support on how to specify ACPs. They broadly classify access control policies into three categories: global ACPs that are built into a system, DAC policies that are basically security requirements, and MAC policies that are essentially mechanisms and should not concern requirements engineering (RE). Our approach is consistent with this view. We specify ACPs from requirements and high-level security and privacy policies.

3. Access control analysis & ACP specification

3.1. Policy hierarchy and software development process

Moffett and Sloman [32] discuss the importance of policy hierarchies and define several relationships (e.g., partitioned targets, goal refinement, delegation of responsibility), which may exist between policies in a hierarchy. We classify policies according to three expression modes: natural language, declarative or semi-structured languages, and formal languages (see Figure 1).

![Figure 1: Policy hierarchy and software development process](image)

Policies at the top level are stated in natural language; examples include website privacy policies, corporate security policies, security and privacy laws, etc. Policies at the middle level are specified in declarative or semi-structured languages, such as Ponder [11] and XACML [14]. These policies instantiate the high-level policies into rules that describe who has permission to access which object in a specific system. There are two major advantages of mid-level policies. First, they are machine-enforceable, whereas natural languages policies are not. Second, it is relatively easier for system security officers to specify security policies that meet organizational security goals using declarative middle-level languages than using formal languages. The policies we derive in this paper fall into the former category. Policies at the bottom level are specified in formal languages such as Authorization Specification Language (ASL) [26] and Alloy [24]. Although formal policies may be directly derived from natural language policies, mid-level policies are helpful for specifying formal policies. It is easier to verify a system’s security properties (which may be automated) using formal policies than using high-level policies.

Moffett treats high-level policies as requirements and low-level policies as an implementation of them [31]. We concur with this view but note additional similarities and differences between policies and requirements. Both policies and requirements express desire or worth rather than fact. Their statements are expressed in the optative mood, specifying what must or ought to be done. On the other hand, policies are more open-ended than requirements in that policies often describe general and high-level goals, whereas requirements are specific, unambiguous and verifiable. The scope of policies is broader than
requirements in that requirements cover one system, whereas policies often govern several systems [1].

We map policies to software development activities as shown in Figure 1. From the software systems’ perspective, high-level security and policy specifications should be specified as security requirements. Mid-level policies are instances of high-level policies within the system’s context. To specify these policies, we must examine system requirements to identify users and their interactions with the system, and system designs (e.g., database design) to identify the data to be protected. Sometimes, these mid-level policies are hard-coded in the system. In other cases, policies are specified by users and enforced by the system, instead of being programmed in the source code.

Our Requirements-based Access Control Analysis and Policy Specification (ReCAPS) approach is based on the general framework discussed above and focuses on specifying ACPs for data-intensive information systems.

3.2. Overview of our approach

Figure 2 portrays ReCAPS using a traditional ICOM (Inputs-Constraints-Outputs-Mechanisms) model. The input sources are: a software requirements specifications (SRS) document, a design document, a DB design (e.g., E/R diagram), and security and privacy policies. The SRS and the DB design are required sources, whereas design document, security and privacy policies are optional. The two required source documents are complementary in that the SRS justifies the rationale for the ACPs (e.g., why a user is given a particular privilege to perform a task or access an object), whereas the database design details the objects to which any access should be controlled.

![Figure 2: ReCAPS ICOM Model](image)

Our approach guides analysts with a detailed process description and heuristics as they examine available source documents and derive ACPs. During the process, analysts employ RE techniques, such as goal [13] and scenario analysis techniques [36], to help identify AC elements. Ambiguities in the SRS are clarified, and inconsistencies between the SRS and database design are identified and resolved. Tool support is currently under development.

The overall objective of our approach is to produce a comprehensive set of ACP specifications. Additionally, the source documents inevitably benefit from the process as the SRS and DB design are augmented, which results in more complete, correct and less ambiguous project documentation. The approach also yields a documented list of design decisions made during the analysis process.

Our approach is based on the following three assumptions that narrow the scope of our efforts to data intensive information systems:

**Assumption #1:** A system’s DB design and SRS exist and are available. Both are pre-requisite source documents for this approach. Essentially, ACPs bridge the gap between an information system’s SRS and its DB design.

**Assumption #2:** There are various objects in a system to which access may be restricted. For example, an employee may be allowed to print jobs to a local printer, but not to a central printer in the neighboring office. This kind of resource access control is beyond the scope of this paper. Instead, we focus on access to data within a DB.

**Assumption #3:** ACPs are specified for information systems, supported by a database containing sensitive data. We have not investigated ACP specifications for security kernels such as file access in operating systems.

Figure 3 portrays the activities an analyst undertakes to derive ACPs from various sources. The process is detailed with examples in Section 4.

![Figure 3: ReCAPS Process Diagram](image)

3.2 Access Control Analysis Principles

The high-level activities described above provide an overview of our method. In practice, we adopt several principles to guide this process.

**Principle #1:** Policies and system requirements must be brought into better alignment to ensure that unauthorized accesses to sensitive data and security breaches are prevented. Software requirements and the operational functioning of policy enforcing systems are often misaligned. The underlying principle of our approach is thus to ensure compliance between access control policies, system requirements and software designs.

**Principle #2:** Traceability between ACPs and requirements must be maintained. Both policies and requirements may change throughout system development and even after the system is deployed. They are interdependent with respect to change. When one of them changes, it is important to make appropriate changes to the other. Traceability helps analysts track changes and maintain consistency between requirements and policies.

**Principle #3:** Access control analysis and requirements analysis is an iterative process. Both require one to maintain documents containing information with different
levels of formality and describing different kinds of information. Therefore, it is helpful to employ an inquiry-driven approach [36] to document important or recurring questions, design decisions, as well as identified inconsistencies that are pending resolution.

4. Case studies
Three case studies enabled us to evaluate the method introduced in Section 3 as we now discuss.

4.1. Security and Privacy Requirements Analysis Tool
The Security and Privacy Requirements Analysis Tool (SPRAT) [25] is a research tool under development at North Carolina State University (NCSU) with funding from the U.S. National Science Foundation (NSF). The tool supports goal-based and scenario-based requirements analysis and provides support for analyzing and specifying security and privacy requirements as well as ACPs. It builds upon and extends two existing tools, the Privacy Goal Management Tool (PGMT) [2] and the Scenario Management and Requirements Tool (SMaRT) [38].

The SPRAT helps analysts seamlessly integrate goals, scenarios, requirements, policies and documents in a project repository and conveniently trace from one element (e.g., a goal) to another (e.g., a scenario) during requirements analysis. The architecture calls for centralized data storage with distributed client access, with an option that allows local data storage. The envisioned users include requirements analysts, policy makers and security engineers. The information stored in the centralized database is proprietary and needs to be protected. Because access control is critical, there are a variety of access control requirements in the system, making it a sufficiently sophisticated system to analyze using our approach.

4.2. Transnational Digital Government (TDG) project
The Transnational Digital Government (TDG) project, funded by the U.S. NSF3, is a collaborative research project involving researchers at seven universities as well as government agencies in three participating countries: U.S., Belize and the Dominican Republic. The project’s objective is to research advanced information technologies useful for rapid collection, dissemination, and exchange of information related to transnational border control. To this end, we are developing a system to collect and share immigration information.

The security and privacy of immigration data is critical in this project. Many organizations are involved in the transnational border control process, including remote border stations, governments, police departments, immigration departments, customs, etc. and information must flow among these organizations as well as across national borders. Different countries have different security and privacy policies and laws regulating the corresponding activities. Policy enforcement was important throughout the system’s development to ensure compliance with the corresponding laws and policies. Using our approach, we specified a set of ACPs for the TDG project.

4.3. Surry Arts Council E-commerce Website
The Surry Arts Council (SAC) is a non-profit organization in Mount Airy, North Carolina that provides opportunities for the local community to experience high-quality performances and instruction in the performing and visual arts. Surryarts.org is the web portal for disseminating information about the Council, such as information about arts events in the community. The SAC contracted a group of students at NCSU to enhance its website with e-commerce functionality. In this study, we had a large amount of documentation available to us: the SRS, DB design, the design document, as well as the organizational security and privacy policies. We analyzed all source documents to derive ACPs for this case study.

4.4. Effort and results
Table 1 summarizes the effort devoted to these case studies and lists the resulting artifacts. During our studies, we observed that it is easier to derive ACPs if the SRS is concrete and detailed. Of the three case studies we analyzed, the TDG SRS is relatively high-level and more ambiguous than the other two projects’ SRS. Thus, some of the AC rules derived are high-level and cannot be mapped to the current DB design. Consider the following TDG requirement:

2.2.2: The system shall allow border immigration agents (BIA) to import data from other agencies into the local database.

This requirement clearly contains access control information. However, we were unable to determine from which agencies data was available and how these data map to the data in the local database. Thus, the rule <BIA, data, import, null, null> derived from this requirement is ambiguous and requires further clarification with stakeholders in Belize and Dominican Republic.

Although the SAC SRS was also somewhat ambiguous, a large number of project documents are available to us. These additional documents helped us clarify many ambiguities as we derived ACPs.

We identified 17 inconsistencies between the SRS and the DB design in the TDG project — a larger number than in the other two studies. Consider the following TDG requirement:

2.3.1: The system shall allow border immigration agents to determine if the traveler is on the “watch list”.

This requirement was annotated with stakeholder comments on what data is contained in the “watch list”. Three of these items were missing in the database schema: gender, the reasons for a person’s name being on the watch list, and actions to be taken if person whose name appears on the watch list is encountered at a border station. It is likely that the distributed nature of this project contributed factored in to this. The SRS was written by the requirements team at NCSU, whereas the DB was designed by the database team at the University of Florida. Insufficient communication between both teams yielded inconsistencies between the requirements and DB design. This is a common software engineering problem and our

3 The TDG Project URL: http://www.acis.ufl.edu/transdg/
case study shows that ReCAPS helps to identify and eliminate these inconsistencies.

<table>
<thead>
<tr>
<th>Table 1. Summary of the three case studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time effort (person-hour)</td>
</tr>
<tr>
<td>No. of functional requirements</td>
</tr>
<tr>
<td>No. (percentage) of new requirements</td>
</tr>
<tr>
<td>No. (percentage) of new requirements created</td>
</tr>
<tr>
<td>No. (percentage) of changed requirements</td>
</tr>
<tr>
<td>No. of access control rules derived</td>
</tr>
<tr>
<td>No. of inconsistencies between the SRS and the DB design</td>
</tr>
</tbody>
</table>

5. Analysis process and heuristics

We now describe the steps an analyst takes to conduct this kind of analysis using examples from the three systems, demonstrating how the heuristics are applied.

Step 1: Understand the scope of access control and the problem domain.

It is important to understand what is inside or outside the scope of access control. If a requirement involves a user’s electronic access to some data in the system, then the user part is inside the scope of access control; otherwise, it is outside the scope. For example, a requirement may state physical security safeguards to control users’ access to the building or system. This is outside the scope of access control. Additionally, when a requirement states a user’s interaction with the system, only the user part is inside the scope of access control.

To obtain a general understanding of the problem domain, analysts need to quickly scan the available source documents. Generally, the SRS contains introductory material, which summarizes the envisioned system, the stakeholders and the end users. It is important to understand the concerns of both end-users and stakeholders to better control end-users’ access to data.

Step 2: Scan the source documents to identify access control elements.

The SRS is the major source from which to derive access control elements. We have created a set of heuristics to help analysts identify the five AC elements: <subject, object, action, condition, obligation>. In the interest of space, we only present a few.

H3: Every object in the SRS should be mapped to an object (e.g., a table, a column, a row or a cell) in the DB design.

Heuristic H3 has two benefits. First, it forces analysts to clearly define what the objects are in the database. Second, it forces analysts to ensure that the requirements and database design are consistent with one another. In all three case studies, H3 was instrumental in helping us identify missing data fields in the database design. Consider the following TDG requirement:

2.3.1: The system shall allow border immigration agents to determine if the traveler is on the “watch list”.

This requirement was annotated with stakeholder comments on what data is contained in the “watch list”. Three of these items were missing in the DB schema: gender, the reasons for a person’s name being on the watch list, and actions to be taken if person whose name appears on the watch list is encountered at a border station. As shown in Table 1, we identified 17 inconsistencies between the TDG SRS and the corresponding DB design. Heuristic H3 allowed us to correct the DB designs early on, preventing possible costly changes that may not have been identified until well into the development lifecycle.

Generally, actors are explicitly stated in the SRS and can be identified intuitively without much difficulty; however, actions are trickier to identify. Actions can be classified into two categories: database actions and abstract actions. Database actions are direct operations to a database, such as insert, update, read, delete, whereas abstract actions are not. For example, withdraw (money from a banking account) and process (orders) are abstract actions. Abstract actions generally map to function-level ACPs. In Step 2, we identify both types of actions and refine them in Step 3.

H4: Given the requirement “The system shall allow <someone> to <do something>”, the actor is <someone> and the action is <do>.

H4.1: When the <do> action is a DB operation, specify that action in the derived AC rule.

H4.2: Actions that are not DB operations must be decomposed into DB operations using scenario analysis.

Consider the following SAC requirement:

FR 1.7: The system shall allow the Surry Arts Council administrators to process orders. This entails viewing, completing, canceling or deleting orders.

The action words in FR 1.7 are process, view, complete, cancel and delete. View and delete are database actions, whereas process, complete, and cancel are abstract actions. To decompose these actions, we must understand the exact interactions between the SAC administrators and the system. By consulting stakeholders, we clarified that “complete/cancel orders” actually means the SAC administrators change the order’s status. Thus, we can decompose the abstract complete action into two database actions: read and update. We document this relationship for further refinement in Step 3.

H7: When a requirement is vague or when actor/action cannot be easily identified, use scenario analysis to elaborate the requirement.

Consider the following ambiguous SPRAT requirement:

FR-PM-3: The system shall support multi-user analyst results comparison.

This requirement was so ambiguous that without clarification it is impossible to understand what it means. Scenario analysis enabled us to identify the main events in this scenario as follows [22]:
Event 1: Analysts classify goals independently according to predefined categories.
Event 2: Project managers select analysts whose classification results they wish to compare.
Event 3: The system shall display those goals that are classified differently and how they are different (e.g., by showing the different categories).

This analysis yields clear actors and actions that are necessary for specifying ACPs.

Defining all the conditions for an access control rule is a challenging task. Although we cannot guarantee these heuristics will identify all conditions, our studies to date show that their use does ensure better coverage.

H9: When requirements specify constraints (e.g., if, unless, when, during, before, after, etc.), the constraints must be specified as conditions.

Consider the following SAC requirement.
FR 1.5: Users can cancel an order before the Surry Arts Council administrators start processing the order.

This requirement explicitly defines a constraint that forbids users to cancel an order. This constraint must be specified as a condition in the access control rule.

H11: Identify conditions by evaluating the rule’s privacy concerns.

There are several sub-heuristics for identifying privacy related conditions with respect to privacy elements such as purpose, recipient, retention, and consent, etc [34]. As previously mentioned, we only specify privacy within the context of privacy. An organization’s privacy policy often states that certain types of data will only be used for a specific purpose (e.g., telemarketing, payment, research and development, etc.). This purpose must be specified as a condition in the access control rule as follows [22]:

\[ \text{action.BusinessPurpose} \ll \text{object.DataPurpose} \]

The symbol \( \ll \) means business purpose is contained in data purpose. Consider the following example: A particular piece of data (e.g., credit card information) is supposed to be used only for payment (data purpose). Given the condition in the ACP rule above, a data access request will be evaluated by an enforcement engine; the business purpose will be checked against the requested object’s data purpose — payment.

H12: Use general security principles such as least privileges and separation of duties to construct misuse cases that a user may exploit the capabilities for hostile intent.

This heuristic is especially helpful to define conditions that are implicit and not easy to identify. Again, consider SPRAT requirement FR-PM-3 mentioned previously. According to FR-PM-3, Project Manager is allowed to view goal classification results. Project Manager and Analysts are not mutually exclusive roles, which means a user can assume both roles at the same time. We thus ask, “can a Project Manager exploit the capability for his/her own good?” and construct the misuse case shown in Figure 4. In this misuse case, user A assumes both roles: Project Manager and Analyst, whereas user B assumes only the Analyst role. The attack pattern is: B classifies goals, then A views B’s results and classifies goals. This is not desirable because A is biased if he sees B’s results before he classifies the goals. Misuse case analysis allowed us to identify a condition for Project Manager to view goal classification results. We document this rule as follows:

\[
\begin{align*}
\text{IF Role (user, Project Manager)} &= \text{TRUE AND} \\
\text{Role (user, Analyst)} &= \text{TRUE AND} \\
\text{user.scheduledToClassify} &= \text{FALSE} \\
\text{user.classifyingFinished} &= \text{FALSE} \\
\text{THEN viewClassificationResults} &= \text{DENY}
\end{align*}
\]

Figure 4: A misuse case to identify conditions

Because traceability is a primary design principle in our approach, analysts document from which requirement the ACP rules were derived. With tool support, the traceability between requirements and access control policies will be maintained more easily.

Step 3: Refine AC rules into ACPs

Recall that an access control policy is comprised of a set of access control rules. The AC rules derived from different requirements could be redundant or in conflict with each other. Thus, we must reconcile the rules identified in Step 2 and specify the collective privileges that to be assigned to each actor. Additionally, how the policies are specified depends on how access control may be implemented. Thus, the refinement process is also a software design process. We sort all the AC rules according to subject and object before we start the refinement process.

An important refinement activity is the reconciliation of synonymous words. In the SPRAT SRS, for example, “allow Analysts to view goal elements” encompasses “allow Analysts to view goal contexts” because context is a goal element. These two rules were merged, yielding: “allow Analysts to view elements of goals”. Another type of redundancy comes from the mode of access control rules. Generally, an allow rule may be specified as a deny rule if the conditions are reversed and vice versa. For example, rule allow A to do B if condition C is satisfied may be specified as deny A to do B if condition C is not satisfied. It is not necessary to have both rules. We may choose a mode to specify an access control rule if that mode is easier to specify the rule. Sometimes combining both modes makes it easier to specify an ACP as shown in the Table 2.

If the requirements from which AC rules were derived are in conflict with one another, the resulting AC rules may conflict as well. Thus, any conflicts must be resolved at this stage. Two types of conflicts may exist:

1. Modality conflicts. In one rule, the subject is allowed to perform an action on an object, but in another rule, the subject is denied to perform the same
action on the same object. Only one of the rules can be true so this type of conflict results from requirements and can only be resolved by consulting stakeholders.

(2) Partial conflicts. For example, in the SAC system, administrators are allowed to view users’ personal information but not their password. Password is part of users’ personal information, but these two objects have different access permissions. There are two ways to resolve the conflict. The first is to decompose users’ personal information and specify several allow rules for each part. Alternatively, both rules can be grouped as a policy. Both rules in the policy must be evaluated during enforcement.

Conflict identification is simplified by analyzing the grouped policies; however, our approach only detects explicit conflicts among the rules. More rigorous conflicts analysis is beyond the scope of this paper.

Analysts often make design decisions while specifying ACPs, resulting in a needed change to some rules. For example, in the SPRAT, we decided to employ RBAC to control users’ access to data. Based on this decision, we merged three privileges for the System Administrator role (create Project Managers, create Analysts and create Guests) into: “create user account” and created a new privilege: “assign roles to users”. It is important to document such design decisions to maintain traceability.

The main artifacts produced by analysts during the ACP specification process are a set of ACPs, documented design decisions, as well as augmented requirements and database design specifications. Table 2 portrays some of the AC policies and rules for the Users role in the SAC project. We will discuss them in detail in Step 4.

**Step 4: From policy specification to software design**

Step 4 entails deciding which policies are implementation-worthy and how to implement them. As previously mentioned, ReCAPS allows software engineers to specify both database-level and function-level access control policies. The policies listed in Table 2 are database-level policies. Each action is a DB operation and each object is an object in the DB. These policies are helpful in software design at least in two ways.

First, they help generate DB SQL queries. For example, rule 5 allows users to read their own purchase records (table payments and orders in the DB). From this rule, we specify the following SQL query:

```sql
SELECT * FROM payments, orders WHERE userid = 111 AND payments.orderstate != cancelled
```

Second, they help software developers during implementation. For example, rule 1 allows users to create a new account. The condition for this rule is *Account does not exist && user provides all required info*. This condition requires software developers to verify whether the user account already exists and whether the required information is provided (and in the right format) when they implement this functionality.

In addition to database-level policies (DLP), ReCAPS also supports function-level policy (FLP) specification. For example, policy 1 contains three AC rules, concerned with user account management. These three rules are derived from the same requirement FR 1.5. In the actual system, these rules are implemented in two functions: (A) one for creating a new account and (B) the other for updating an existing account. If software developers implement these three rules using function-level access control, these three rules can be refined into the following two policies:

- **FLP 1**: Allow user to access function A
- **FLP 2**: Allow user to access function B if user is logged in

The two policies result in fewer conditions to be checked during run time, making them more efficient and easier to implement. Basically, FLP 1 becomes unnecessary as an ACP, given that anyone can access function A (i.e., create a new account). FLP 2 requires a

<table>
<thead>
<tr>
<th>Policy No</th>
<th>Rule No</th>
<th>Mode</th>
<th>Subject</th>
<th>Action</th>
<th>Object</th>
<th>Condition</th>
<th>Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Allow</td>
<td>Role (User)</td>
<td>Insert</td>
<td>Useracct</td>
<td>Account does not exist &amp;&amp; user provides all required info</td>
<td>FR 1.5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Allow</td>
<td>Role (User)</td>
<td>Read</td>
<td>Useracct</td>
<td>User’s account exists &amp;&amp; user is logged in &amp;&amp; only user’s own account info</td>
<td>FR 1.5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Allow</td>
<td>Role (User)</td>
<td>Update</td>
<td>Useracct</td>
<td>User’s account exists &amp;&amp; user is logged in &amp;&amp; only user’s own account info &amp;&amp; all fields meet restrictions</td>
<td>FR 1.5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Allow</td>
<td>Role (User)</td>
<td>Insert</td>
<td>Payments</td>
<td>User provides all required info</td>
<td>FR 1.1-1.4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>Allow</td>
<td>Role (User)</td>
<td>Read</td>
<td>Payments</td>
<td>User is logged in &amp;&amp; the requested order exists &amp;&amp; only user’s own order and payment info &amp;&amp; order status not cancelled</td>
<td>FR 1.5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>Deny</td>
<td>Role (User)</td>
<td>Update</td>
<td>payments.orderstate</td>
<td>The requested order has been started processing</td>
<td>FR 1.5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>Allow</td>
<td>Role (User)</td>
<td>Update</td>
<td>payments.orderstate</td>
<td>User is logged in &amp;&amp; the requested order exists &amp;&amp; only user’s own order and payment info</td>
<td>FR 1.5</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>Allow</td>
<td>Role (User)</td>
<td>Insert</td>
<td>Orders</td>
<td>User provides all required info</td>
<td>FR 1.1-1.4</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>Allow</td>
<td>Role (User)</td>
<td>Read</td>
<td>Orders</td>
<td>User is logged in &amp;&amp; the requested order exists &amp;&amp; only user’s own order and payment info &amp;&amp; order status not cancelled</td>
<td>FR 1.5</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>Allow</td>
<td>Role (User)</td>
<td>Read</td>
<td>Catalog</td>
<td></td>
<td>FR 1.1-1.4</td>
</tr>
</tbody>
</table>
user to log in before accessing function B (i.e., access his/her account information), but now the other conditions of access control (e.g., user can access only his/her own account info) are encapsulated within function B.

6. Discussion and plans for future work
ACP specification is typically conducted without methodological support or systematic guidance. ReCAPS supports access control analysis and ACP specification by providing prescriptive guidance with a rich set of heuristics that help requirements and software engineers specify access control policies. ReCAPS thus offers three main advantages not currently available. Specifically it offers: prescriptive guidance using RE techniques for ACP specification, traceability support, and improvements to requirements specifications and software designs.

The traceability support between requirements, ACPs and designs is a prominent ReCAPS feature. Figure 5 portrays the traceability links across various artifacts. When an ACP is derived, we establish a link between the policy and its sources (e.g., a particular requirement, a particular section of the security policy or privacy policy, or a particular table/field in the database design). When we employ RE techniques to elaborate requirements, we also build requirements traceability links between each requirement and its origin, goals, scenarios and stakeholders. By establishing the links across these artifacts, we can manage policy and/or requirements evolution. In the event of a change in a policy or requirement, our approach allows analysts to quickly locate the affected requirements or policies for subsequent modification. By ensuring consistency between ACP, requirements and software designs, our approach improves the quality of ACPs and helps bridge the gap between requirements and design.

![Figure 5: Traceability support in ReCAPS](image)

Because ReCAPS is inherently inquiry-driven and iterative, it offers the same benefit as other inquiry-driven approaches. As such, it helps analysts ensure that many of the ambiguities, inconsistencies, and conflicts that often plague requirements specifications, software designs and corresponding policies are identified and eliminated.

In addition to deriving ACPs from requirements and DB designs, we have validated ReCAPS’ usefulness for deriving ACPs from other source documents as well. In the SAC project, for example, we applied the heuristics to derive ACPs from high-level security/privacy policies. Moreover, ReCAPS provides support for both database-level and function-level ACP specification.

We have conducted a pilot empirical study in a graduate-level software engineering course to evaluate ReCAPS’ effectiveness and usefulness. Although the sample size was relatively small, the results provide preliminary anecdotal evidence that applying the ReCAPS yields better results than ad-hoc ACP specification. The pilot study enabled us to refine our empirical instrumentation, which is presently in use in another empirical study to further evaluate the method.

Our approach does have its limitations. To date, we have applied ReCAPS in projects for which an initial database design was available. Ideally database design should be driven by security requirements. We plan to investigate situations when database designs are not available and how ReCAPS may help database design. Additionally, An AC rule can have various modes (e.g. permit/deny/oblige/refrain). In our studies, we have found allow/deny rules to be sufficient for the systems we have evaluated. We plan to explore the efficacy of other modes to possibly extend our approach accordingly.

Ensuring a set of policies is complete and consistent is a major challenge [26]. ReCAPS offers a first step towards this objective. The techniques to analyze the completeness and consistency of software requirements are applicable to ACP analysis and we plan to investigate this further in our future work.

To date, we have validated our approach within the context of three systems. All three studies, involving real systems, demonstrate the applicability of ReCAPS for specifying ACPs that are consistent with system requirements and high-level policies. We intend to continue our evaluations of the approach’s overall effectiveness and investigate useful extensions, for example, extending ReCAPS to define roles for RBAC systems [22].

Finally, legal requirements play an important role in protecting consumer data security and privacy (e.g., the HIPAA, COPPA, and GLBA in the U.S.). These kinds of documents are inherently different from the kinds of documents we have examined to date. Ensuring requirements and policies are in compliance with security and privacy laws remains a challenge that we will continue to investigate within the context of the TDG project.

Acknowledgements
This work was supported by NSF Digital Government Grant #0131886 and ITR Grant #0325269. We thank the TDG project team for their collaboration as well as Calvin Powers, Jonathan Moffett, Ting Yu and the NCSU ThePrivacyPlace.Org reading group for their comments.

References


