RC24056 (W0609-065) September 19, 2006
Computer Science

IBM Research Report

When Role Models Have Flaws:
Static Validation of Enterprise Security Policies

Marco Pistoia, Stephen J. Fink, Robert J. Flynn*, Eran Yahav
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

*Polytechnic University
Brooklyn, NY

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

When Role Models Have Flaws: Static Validation of Enterprise Security Policies

Marco Pistoia Stephen J. Fink
IBM Watson Research Center
Hawthorne, New York
{pistoia,sjfink}@us.ibm.com

Robert J. Flynn
Polytechnic University
Brooklyn, New York
flynn@poly.edu

Eran Yahav
IBM Watson Research Center
Hawthorne, New York
eyahav@us.ibm.com

Abstract

Modern multiuser software systems have adopted Role-Based Access Control (RBAC) for authentication and authorization management. This paper presents a formal model for RBAC policy validation and a static analysis model for RBAC systems that can be used to (1) identify the roles required by users to execute an enterprise application, (2) detect potential inconsistencies caused by principal delegation policies, which are used to override a user's role assignment, (3) report if the roles assigned to a user by a given policy are redundant or insufficient, and (4) report vulnerabilities which can result from unchecked intra-component accesses. The algorithms described in this paper have been implemented as part of IBM's Enterprise Security Policy Evaluator (ESPE) tool. Experimental results show that the tool found numerous policy flaws, including ten previously unknown flaws from two production-level applications, with no false positive reports.

1 Introduction: RBAC Systems

Role-Based Access Control (RBAC) [9] has become a popular authorization model for managing enterprise-scale applications. Many enterprise software systems support RBAC, including Java, Enterprise Edition (EE)[1] [21], Microsoft .NET Common Language Runtime (CLR) [10], and modern database management systems.

An RBAC policy restricts access to protected operations based on "roles". A *role* is a semantic grouping of access rights, which can be assigned to users and groups of an application or system.

Typically, a system administrator manages the RBAC policy via declarative artifacts, distinct from the application code. Unfortunately, the code and the security policy can interact in complex and subtle ways, based on possible component dependencies, principal delegation policies, and the run-time authorization model. In this complexity, an RBAC policy may hide security vulnerabilities, which can be extremely difficult to find with testing or code inspection.

To alleviate these problems, we present a static analysis model to verify that an RBAC policy does not exhibit certain classes of vulnerabilities. This paper presents a novel theoretical foundation and an interprocedural analysis framework to model the flow of authorization information in an RBAC system and automatically detect security policy misconfigurations.

The model presented in this paper identifies three types of RBAC security flaws, corresponding to RBAC policies that are:

- *Insufficient*, leading to stability problems due to potential run-time authorization failures,
- *Redundant*, granting a superset of the minimal set of roles necessary to execute a program, thus violating the Principle of Least Privilege [23], or
- *Subversive*, permitting an execution to bypass declared access restrictions by exploiting unchecked intra-component calls.

We present a sound static analysis algorithm to verify that an RBAC policy is sufficient and not subversive, and a complete analysis to identify redundant policies. The analyses can also suggest alternative policies which remedy these flaws.

We have implemented the algorithms described in this paper as part of an IBM tool called Enterprise Security Policy Evaluator (ESPE). We present the results obtained by executing ESPE on a number of Java EE applications. The tool found numerous security policy flaws, including previously unknown flaws from publicly available codes and from two production-level commercial applications, with no false positive reports.

[1] Formerly known as Java 2, Enterprise Edition (J2EE).

2 Motivating Example

Typically, an RBAC system allows any resource to be restricted with zero, one, or multiple roles. If no role protects a resource, any principal may access it. When multiple roles restrict a resource, the role requirement manifests as a logical `OR`; a principal must possess at least one of the specified roles. If multiple resources access each other, forming a chain of calls, then the user accessing the first resource in the chain needs roles authorizing each link in the chain; the role requirement manifests as a logical `AND`. Thus, in order to configure an RBAC policy correctly, a system administrator must infer which role requirements could arise in any possible execution, and evaluate potentially complex logical expressions of these role requirements.

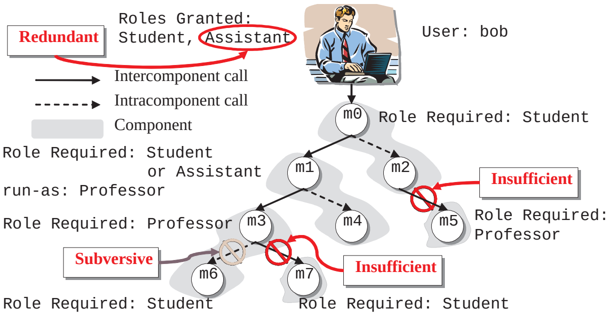

Figure 1. RBAC in Component-Based System

Figure 1 shows a simple example of an RBAC policy for a distributed component-based system, such as Java EE. The component run-time system (container) intercepts inter-component method calls, and checks RBAC authorization at that time. In Figure 1, user `bob` has been granted the `Student` role, which satisfies the role requirement to invoke entry method `m0`.

Unfortunately, the behavior of `m0` will lead to an authorization failure downstream. The call graph shows that `m0` leads to the invocation of an inter-component call from `m2` to `m5`, which has been access-restricted with the `Professor` role. This invocation will fail at runtime because `bob` does not have the required role.

In practice, the container will often fail to maintain enough information to report useful error messages. In Java EE, the container will report the following unhelpful error message at run-time:

```
java.rmi.ServerException:
Nested exception is: java.rmi.AccessException:
CORBA NO_PERMISSION 9998
```

For large applications, manually tracking back the error across the distributed call stack presents a difficult challenge. For this reason, it is desirable for authorization failures to occur immediately on entry to an application. We call a policy that does not enforce this desirable property *insufficient*.

The example also violates the Principle of Least Privilege [23], which states that a user should be granted no more privileges than necessary. The policyr grants `bob` the role `Assistant`, which is extraneous for the example code. Note that even though a call to `m1` would be satisfied by the `Assistant` role, since the `Student` role is required anyway, granting the `Assistant` role is unnecessary. A *redundant* policy, which grants unnecessary privileges, can lead to security problems as the code evolves over an application's lifetime.

2.1 Principal Delegation

In an RBAC system, the identity of the principal who initiates a transaction propagates to downstream calls. However, some resources may need execute as though called by a principal with different, perhaps more privileged, roles. For this purpose, most RBAC systems allow the administrator to map each component to a *principal delegation policy* and override the identity of the executing principal with a specified identity. At runtime, all the downstream calls from that point on assume the roles held by that identity.

In the example, the component of `m1` uses a `run-as` delegation policy that forces all the subsequent downstream calls to be performed under the `Professor` role. Thanks to this delegation policy, the authorization check for `m3` succeeds.

Principal delegation polices can lead to authorization failures. In the example, the component of `m3` does not set the principal delegation policy back to `Student`, which is the role requirement to invoke `m7`. Thus, `bob`, who was granted the `Student` role at the beginning, is now denied access to `m7` for not having the role of `Student`. In this case, the principal delegation has, as a side effect, made the RBAC policy insufficient.

2.2 Intra- vs. Inter-component Calls

Typically component-based systems enforce authorization checks only across component boundaries. Once an execution enters a component, the system does not perform authorization checks for further behavior inside said component [30]. This design favors execution performance. However, if the access-control policy for an internal execution point differs from that of the component entry point, unintended security violations can occur.

In the example, the intra-component call from `m3` to `m6` will *not* fail, even though when executing `m3` the principal does not hold the required role, `Student`. In general, when an execution can bypass declared authorization checks by exploiting intra-component calls, we call the execution *subversive*.

3 A Formal Model for RBAC

This section presents a formal model for RBAC and defines the notions of sufficiency, minimality, and subversion.

3.1 Concrete Semantics

Given a program p with sets of methods M and roles R, we define *role formulae* to be propositional logic statements over R, where each $r \in R$ is considered as a predicate. We denote the set of role formulae over R by $\mathcal{B}(R)$.

Definition 3.1. *An* RBAC policy *for p is a tuple $P = (R, U, \upsilon, \mu, \pi)$, where*

1. *R is a finite set of* role predicates.
2. *U is a finite set of* users.
3. *$\upsilon : U \to \mathcal{B}(R)$ is the* user role assignment function*, interpreted as follows: $\forall u \in U$, $\upsilon(u)$ defines the roles granted to u at program entry; $\upsilon(u) \not\Leftrightarrow \texttt{false}$.*
4. *$\mu : M \to \mathcal{B}(R)$ is the* role requirement function*, mapping each method to the roles required to invoke it.*
5. *$\pi : M \rightharpoonup \mathcal{B}(R)$ is the* principal delegation partial function*, defined as follows: $\forall m \in M$, $\pi(m)$, if defined, indicates the roles that m sets as part of its principal delegation policy; $\pi(m) \not\Leftrightarrow \texttt{false}$.*

Function μ typically requires a disjunction of roles, allowing a user u to invoke a method m if and only if u possesses any of the roles in the disjunction $\mu(m)$. More precisely, if $g \in \mathcal{B}(R)$ defines the roles dynamically held by an execution when it invokes m, the invocation of m will succeed if and only if $g \Rightarrow \mu(m)$. In particular, if $\mu(m) \Leftrightarrow \texttt{true}$, then m is unprotected, while if $\mu(m) \Leftrightarrow \texttt{false}$, then m is inaccessible.

Function υ typically assigns a conjunction of roles, granting a user multiple roles simultaneously. If $\upsilon(u) \Leftrightarrow \texttt{true}$ for some user u, then u can only access unprotected resources.

The π partial function models changes in privileges according to principal delegation policies. If, for a method m, $\pi(m)$ is defined, then when the program makes a call from m, it assumes roles $\pi(m)$. When $\pi(m)$ is not defined, it signifies *no* principal delegation in place; the roles held by the user executing p do not change. If $\pi(m) \Leftrightarrow \texttt{true}$, the principal delegation policy strips all privileges; in this state, the program can access only unprotected resources.

We now define, informally, an instrumented concrete semantics to describe the behavior of a program under an RBAC policy. We assume a standard concrete semantics for a program in the underlying language, where the program state consists of a program counter, stack, heap, local variables, and global variables. We assume for this discussion a single thread of execution; generalizing to multiple threads does not introduce any difficulties. We instrument the program state additionally with a stack w of dynamically held roles; if S is the program configuration under the standard concrete semantics, then $\langle S, w\rangle$ is the program configuration under the instrumented concrete semantics. The stack alphabet is $\Sigma := \mathcal{B}(R)$; each $\sigma \in \Sigma$ represents roles that an execution may hold at a particular point.

Definition 3.2. *The* base instrumentation *for an execution initiated by a user u is defined as follows. Given a configuration $\langle S, w\rangle$, we denote a transition of the instrumented concrete semantics into a configuration $\langle S', w'\rangle$ by $\langle S, w\rangle \Rrightarrow \langle S', w'\rangle$. Since the only operations that affect the instrumentation are method calls and returns, we only describe the effect of these operations. When a security violation occurs, the semantics transitions into a designated authorization error state. In the following, we only show transitions to non-error states (implicitly defining all other transitions as transitions to the error state), and we assume that S' is the updated configuration according to the standard concrete semantics applied to S.*

- Init*: Call to entry point m'*

$$\langle S, \epsilon\rangle \Rrightarrow \langle S', \upsilon(u)\epsilon\rangle, \quad \upsilon(u) \Rightarrow \mu(m')$$

- Call*: m calls m'*

$$\langle S, \sigma w\rangle \Rrightarrow \begin{cases} \langle S', \sigma\sigma w\rangle, & \pi(m) \text{ undefined } \wedge \sigma \Rightarrow \mu(m') \\ \langle S', \pi(m)\sigma w\rangle, & \pi(m) \text{ defined } \wedge \pi(m) \Rightarrow \mu(m') \end{cases}$$

- Return

$$\langle S, \sigma w\rangle \Rrightarrow \langle S', w\rangle$$

3.2 Accounting for Intra-component Calls

As discussed in Section 2.2, an RBAC system typically performs authorization checks only on inter-component calls. This unintuitive semantics can lead to unexpected violations of the Principle of Least Privilege, if not accounted for correctly. We now extend the instrumented concrete semantics to distinguish between intra-component and inter-component calls. We denote the set of modules of p by MD and define a function md: $M \to$ MD that maps each method to its module. The modified semantics is identical to those of Section 3.1, with the following special rule, which overrides the semantics of intra-component procedure calls:

Definition 3.3. *The* modified instrumentation *for an execution is identical to the base instrumentation for inits and returns, and changes the handling of calls as follows:*

- Intercomponent Call*: m calls m',* $\mathrm{md}(m) \neq \mathrm{md}(m')$

$$\langle S, \sigma w\rangle \Rrightarrow \begin{cases} \langle S', \sigma\sigma w\rangle, & \pi(m) \text{ undefined } \wedge \sigma \Rightarrow \mu(m') \\ \langle S', \pi(m) w\rangle, & \pi(m) \text{ defined } \wedge \pi(m) \Rightarrow \mu(m') \end{cases}$$

- Intra-component Call*: m calls m',* $\mathrm{md}(m) = \mathrm{md}(m')$

$$\langle S, \sigma w\rangle \Rrightarrow \langle S', \sigma\sigma w\rangle$$

Note that the rule for intra-component calls does not have any side condition. Therefore such calls cannot lead to an authorization check error.

3.3 Sufficiency, Minimality, Subversion

This section formalizes the notion of an RBAC policy being too restrictive or too permissive.

Definition 3.4. *An RBAC policy P for a program p is* sufficient *if for any user u and for any execution e such that $\upsilon(u) \Rightarrow \mu(m_e)$, where $m_e \in M$ is the entry point of e, e does not transition to an authorization error state;* insufficient *otherwise.*

In other words, for P to be sufficient, it is necessary that no execution transitions to an authorization error state, provided the user has permission to initiate the execution by calling its entry point. If a security authorization failure occurs, it must occur immediately when the user calls an entry point (for example, an application service).

It is possible to define a partial order on the class of all the RBAC policies on a program p that share the same role requirement function.

Definition 3.5. *Given a program p with sets of methods M, users U, and roles R, we define a partial order on the class $\mathcal{C}(U, R, \mu)$ of all the RBAC policies on p sharing the same role requirement function $\mu : M \rightarrow \mathcal{B}(R)$, as follows.*

Given two user role assignment functions $\upsilon_1, \upsilon_2 : U \rightarrow \mathcal{B}(R)$, we say that υ_1 is less permissive *than υ_2, and write $\upsilon_1 \preceq \upsilon_2$, if $\upsilon_1(u) \Rightarrow \upsilon_2(u), \forall u \in U$. If $\upsilon_1 \preceq \upsilon_2 \wedge \exists u \in U : \upsilon_2(u) \not\Rightarrow \upsilon_1(u)$, we say that υ_1 is* strictly less permissive *than υ_2, and write $\upsilon_1 \prec \upsilon_2$.*

Given two principal delegation partial functions $\pi_1, \pi_2 : M \rightharpoonup \mathcal{B}(R)$, we say that π_1 is less permissive *than π_2, and write $\pi_1 \preceq \pi_2$, if $\forall m \in M$ either $\pi_1(m) \Rightarrow \pi_2(m)$ or both $\pi_1(m)$ and $\pi_2(m)$ are undefined. If $\pi_1 \preceq \pi_2$, and $\exists m \in M : \pi_2(m) \not\Rightarrow \pi_1(m)$, we say that π_1 is* strictly less permissive *than π_2, and write $\pi_1 \prec \pi_2$.*

Given two RBAC policies $P_1 = (R, U, \upsilon_1, \mu, \pi_1), P_2 = (R, U, \upsilon_2, \mu, \pi_2) \in \mathcal{C}(U, R, \mu)$, we say that P_1 is less permissive *than P_2, and write $P_1 \preceq P_2$ if $\upsilon_1 \preceq \upsilon_2$ and $\pi_1 \preceq \pi_2$. If $P_1 \preceq P_2$ and $\upsilon_1 \prec \upsilon_2$ or $\pi_1 \prec \pi_2$, we say that P_1 is* strictly less permissive *than P_2, and write $P_1 \prec P_2$.*

Intuitively, if $P_1 \prec P_2$, then P_1 is *stricter* than P_2; P_1 grants fewer privileges to the users executing the code. This allows reasoning about the Principle of Least Privilege; an RBAC policy should grant the minimum set of privileges necessary to prevent authorization failures. More formally, an RBAC policy should be "minimal":

Definition 3.6. *An RBAC policy P sufficient for a program p is* minimal *if there exists no sufficient RBAC policy Q for p such that $Q \prec P$; otherwise, P is* redundant*.*

Given a program, it is possible to execute it with respect to either the base or the modified instrumentation. In the latter case, the execution will differ in the state of roles verified and granted, and in possible transitions to authorization error states.

Definition 3.7. *An RBAC policy P is* subversive *if there exists any execution with P that transitions to an authorization error state under the base instrumentation, but not under the modified instrumentation.*

In other words, P is *subversive* if sufficient under the modified instrumentation, but insufficient under the base instrumentation. From a security perspective, a subversive execution is a violation of the Principle of Least Privilege because a user may access a restricted resource, bypassing the intended security policy by exploiting unchecked intra-component calls.

4 Static Analysis for RBAC

Let P be an RBAC policy for a program p. At each program point, it is important to detect whether P is either too permissive or too restrictive. To do this, it is important to identify: (i) the set of roles required at each program point, and (ii) the set of roles the user may possess at each program point, based on the concrete semantics of P. This section describes a conservative Role Requirement Analysis that approximates these sets, and shows how this analysis can identify RBAC policies that can cause security flaws. The Role Requirement Analysis can be used to infer alternative policies that correct such flaws.

4.1 Role Domain

At each point during execution, the RBAC policy requires a certain set of roles, and the user holds a certain set of roles. The Role Requirement Analysis computes values that overapproximate the set of roles required at any program point. The values computed are preconditions that ensure that a call to a given method m will not lead to authorization errors, including (transitively) any further calls from m.

As described in Section 3.1, an RBAC policy specifies role requirements and user authorization in terms of role formulae over the set of roles R, $\mathcal{B}(R)$. Henceforth, we assume that these role formulae are *monotone* (no negations), and specified in conjunctive normal form.

We present the Role Requirement Analysis as a dataflow analysis over sets of roles. To this end, we map role formulae in monotone conjunctive normal form (MCNF(R)) to elements of $\mathcal{P}(\mathcal{P}(R))$ via a mapping ϕ: For $\sigma \in$ MCNF(R), with $\sigma = \bigwedge_{i=1}^{k} \bigvee_{j=1}^{t_i} r_{ij}$,

$$\phi(\sigma) := \{R_i\}_{i=1}^{k}, \text{ where } R_i := \{r_{ij}\}_{j=1}^{t_i}$$

Via this mapping, we proceed to define a set-based dataflow analysis over the semilattice $\mathcal{L}_{\mathcal{S}}$ with elements $\mathcal{P}(\mathcal{P}(R))$, join operator $\cup$, and partial order $\supseteq$. Appendix A shows that, with appropriate quotients, this semilattice is isomorphic through ϕ to the corresponding semilattice derived from MCNF(R) with join operator $\wedge$ and partial order $\Rightarrow$. This result ensures that the set-based dataflow analysis formulation is a faithful representation of the analysis problem derived from the concrete semantics defined in terms $\mathcal{B}(R)$ and $\Rightarrow$.

4.2 Role Requirement Analysis

The first step of the Role Requirement Analysis algorithm is to build a call graph $G = (N, E)$ overapproximating the method calls during execution of the application, and to compute the RBAC policy P by identifying which methods in the application are access-restricted and which components define principal delegation policies. If a method is reachable from the program entry point, then it will correspond to a node in G.

The Role Requirement Analysis overapproximates the sets of roles required at each program point via the solution to a backwards dataflow problem induced by G and P. We first define $m_c \colon N \to M$, mapping a call graph node to its corresponding method. Functions $\mu_c, \pi_c \colon N \to \mathcal{P}(\mathcal{P}(R))$ and $\mathrm{md}_c \colon N \to \mathrm{MD}$ are defined as follows: $\mu_c(n) := \mu(m_c(n)), \pi_c(n) := \pi(m_c(n))$, and $\mathrm{md}_c(n) = \mathrm{md}(m_c(n)), \forall n \in N$. Dataflow functions Gen, Kill : $E \to \mathcal{P}(\mathcal{P}(R))$ are defined as follows, $\forall e = (n_1, n_2) \in E$:

$$\mathrm{Gen}(e) := \begin{cases} \mu_c(n_2), & \mathrm{md}_c(n_1) \neq \mathrm{md}_c(n_2) \wedge \pi_c(n_1) \text{ undef} \\ \varnothing, & \text{otherwise} \end{cases}$$

$$\mathrm{Kill}(e) := \begin{cases} \mathcal{P}(R), & \mathrm{md}_c(n_1) \neq \mathrm{md}_c(n_2) \wedge \pi_c(n_1) \text{ defined} \\ \varnothing, & \text{otherwise} \end{cases}$$

We use Gen and Kill in the following dataflow equations:

$$\mathrm{Out}(e) = (\mathrm{In}(e) \setminus \mathrm{Kill}(e)) \cup \mathrm{Gen}(e)$$
$$\mathrm{In}(e) = \textstyle\bigcup_{f \in \Gamma^+(e)} \mathrm{Out}(f)$$

where $\Gamma^+((n_1, n_2)) := \{(n_1', n_2') \in E : n_1' = n_2\}$.

These monotone dataflow equations are solved via iteration to a fixed point [1]. In the solution, Out$((n_1, n_2))$ overapproximates the roles required for a user invoking $m_c(n_2)$ from $m_c(n_1)$. We define a function $\Lambda : N \to \mathcal{B}(R)$ that annotates each node with (an overapproximation of) the transitive role requirement needed to call that node from another component. $\Lambda(n)$ is defined as follows, $\forall n \in N$:

$$\Lambda(n) = \phi^{-1}((\bigcup_{(n,n') \in E} \mathrm{Out}(n, n')) \cup \mu_c(n))$$

4.3 Security Analyses

The Role Requirement Analysis forms the basis for several security analyses. Specifically, given an RBAC policy $P = (R, U, \upsilon, \mu, \pi)$ as in Definition 3.1, the Role Requirement Analysis can be used to automatically detect if P is insufficient, minimal, or subversive. If it detects that P is insufficient or subversive, the analysis can report code locations that make P insufficient or subversive, and output alternative υ and π that can make P sufficient or non-subversive, respectively. If it detects that P is redundant, the analysis can report a more accurate policy P in which, according to the analysis, the redundancy has been eliminated.

4.3.1 Sufficiency Analysis

This section defines a notion of an RBAC policy's being sufficient with respect to the Role Requirement Analysis result, and then proves a related soundness property.

We define the following two types of edges in the call graph:

- **Entry Edges.** We augment the call graph $G = (N, E)$ with an additional node $\tilde{n} \in N$ representing a generic client invoking p's entry points. If $m_0 \in M$ is an entry point of p and $n_0 \in N$ is a call graph node representing m_0, then the edge $e_0 := (\tilde{n}, n_0) \in E$ is called an *entry edge*.
- **`run-as` Edges.** An edge $e = (n_1, n_2) \in E$ is said to be a `run-as` *edge* if it is an inter-component edge and if $\pi_c(n_1)$ is defined. Run-as edges are exactly those on which the Kill function is nontrivial.

For notational convenience in later sections, we define the result of sufficiency analysis as indicating a policy to be "abstractly sufficient:"

Definition 4.1. *Let P be an RBAC policy and $\Lambda : N \to \mathcal{B}(R)$ the result of Role Requirement Analysis for P. P is* abstractly sufficient *if the following two conditions hold:*

1. *For each entry edge $e_0 = (\tilde{n}, n_0) \in E$, $\upsilon(u) \Rightarrow \Lambda(n_0)$ for each $u \in U$ such that $\upsilon(u) \Rightarrow \mu(n_0)$.*
2. $\pi(n_1) \Rightarrow \Lambda(n_2)$ *for each* `run-as` *edge* $(n_1, n_2) \in E$.

Otherwise, P is said to be abstractly insufficient.

It is straightforward to verify whether an RBAC policy P is abstractly sufficient based on Definition 4.1. The following theorem establishes soundness of sufficiency analysis as so defined:

Theorem 4.1. *If an RBAC policy P is abstractly sufficient, then it is sufficient.*

Proof. Straightforward by induction on the structure of the call graph. □

Of course, there can be RBAC policies that are abstractly insufficient, but sufficient in the concrete semantics. These false alarms can arise due to overapproximation in the call graph on which the analysis relies.

4.3.2 Minimality Analysis

Using the results of sufficiency analysis, we now present an analysis to determine if an RBAC policy is redundant, violating the Principle of Least Privilege.

Given a Role Requirement Analysis Λ for an RBAC policy P, a simple greedy algorithm can search for an alternative policy Q such that P is more permissive than Q, but Q is still sufficient. This minimality analysis iteratively removes one role from role assignments $\upsilon(u), \forall u \in U$, and $\pi(m), \forall m \in M$, and verifies via Λ whether the resulting RBAC policy is still abstractly sufficient. This process terminates after at most $\mathcal{O}(|R|(|E|+|M|))$ iterations and leads to a sufficient RBAC policy that better satisfies the Principle of Least Privilege.

The following result shows that this minimality analysis algorithm is *complete*:

Corollary 4.1. *If an RBAC policy P is abstractly sufficient, and there exists an abstractly sufficient policy Q such that $Q \prec P$, then P is redundant.*

Proof. By Theorem 4.1, both P and Q are sufficient. Therefore, according to Definition 3.6, P is redundant. □

Thus any problems reported by minimality analysis represent actual violations of the Principle of Least Privilege; there are no false alarms. The analysis is not sound, and may fail to identify some violations.

4.3.3 Subversion Analysis

It is easy to construct a Role Requirement Analysis that overapproximates the roles required by a program p assuming that p will be executed with respect to the base instrumented semantics as explained in Definition 3.2. To do so, it is enough to modify the dataflow computation presented in Section 4.2 by disregarding the distinction between inter- and intra-component edges, and considering all calls as inter-component calls. Technically, we define each call graph node to be in a separate module by using a modified set of modules MD' and a module mapping function $\mathrm{md}'_c \colon N \to \mathrm{MD}'$.

We define a labeling function Λ', that annotates the call graph with the solution of the dataflow equation under the module mapping md'_c.

The notions of an RBAC policy's being *abstractly sufficient* or *abstractly insufficient with respect to the base instrumented semantics abstraction* can be obtained from Definition 4.1 by simply replacing Λ with Λ'. For simplicity, we will say that P is *abstractly sufficient for* Λ [Λ'] if is abstractly sufficient with respect to the modified [base] instrumented semantics abstraction; *abstractly insufficient for* Λ [Λ'] otherwise.

It is easy to see that Theorem 4.1 applies to the base instrumented semantics as well, with straightforward modifications. In particular, this means that an RBAC policy P that is abstractly sufficient for Λ' is sufficient with respect to the base instrumented semantics. Thus the subversion analysis is sound:

Corollary 4.2. *If a RBAC policy P is abstractly sufficient for Λ', then it is not subversive.*

With Corollary 4.2, the analysis can verify that a policy is not subversive, with the analysis subject to potential false alarms.

5 Implementation

This section describes the implementation of the security analyses described in Section 4.3.

5.1 General Architecture

ESPE runs as a stand-alone application, which can be launched on top of a Java, Standard Edition (SE) run-time environment from the command line or from Eclipse [8]. It can analyze Java EE deployed applications packaged in Enterprise ARchive (EAR) files or separate Java ARchive (JAR) and Web ARchive (WAR) files [30]. These files contain the object code and deployment descriptors of one or more applications. Source code is unnecessary since ESPE analyzes object code. ESPE analyzes standard Java EE applications regardless of the platform vendor. All the library files used by the applications at run time must be part of the analysis to allow ESPE to provide sound results.

ESPE comprises two main components: a deployment descriptor analyzer and a security analysis engine. ESPE analyzes the object code and produces a call graph modeling the execution of the applications. ESPE analyzes also the deployment descriptors to detect which resources have been access-restricted with roles, and which components define principal delegation policies. Based on this information, the deployment descriptor analyzer produces two mappings: one mapping associates enterprise resources with the roles necessary to access them; the other mapping associates each component with the `run-as` role specified by that component, if any. Next, ESPE analyzes both the call graph and the security mappings, and identifies security and stability problems.

After the call graph has been built, as explained in Section 5.2, the ESPE engine performs the Role Requirement Analysis. Section 6 presents the experimental results obtained by executing ESPE on a number of applications.

5.2 Call Graph Construction

ESPE's security analysis engine relies on a Java EE bytecode analysis system called DOMO, developed at IBM Research. DOMO provides a range of call graph construction algorithms, ranging from class hierarchy analysis [6] to control-flow analysis with a variety of context-sensitivity policies [11]. With these choices, ESPE supports a range of cost/precision trade-offs.

A crucial implementation challenge for Java EE analysis concerns accurate modeling of inter-component calls. Consider for example an enterprise bean having remote interface `Bean2`, remote home interface `Bean2Home`, and enterprise bean class `Bean2Bean` [27]. Suppose that method `m1` in enterprise bean `Bean1Bean` calls remote method `m2` on `Bean2Bean`. For this to be possible, `m2` must be a method declared in `Bean2` and implemented in `Bean2Bean`, and a code similar to the following must be embedded in `m1`.

```
Context initial = new InitialContext();
Object objref =
   initial.lookup("java:comp/env/ejb/Bean2");
Bean2Home bean2Home = (Bean2Home)
   PortableRemoteObject.narrow(
   objref, Bean2Home.class);
Bean2 bean2Object = bean2Home.create();
bean2Object.m2();
```

At the bytecode level, the call to `bean2Object.m2()` delegates to an implementation generated automatically by the Java EE deployment tool. This implementation would consult run-time registries and pass a message over Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) to the process hosting the home container for the bean. The receiving process would unmarshall the RMI-IIOP message, activate the relevant component through the bean lifecycle implementation, and finally delegate to a reflective call to complete the remote invocation.

Analyzing bytecode solely, it would be impossible to resolve this remote method invocation, since the relevant dispatch tables are effectively encoded in the XML deployment descriptor, and read by the container at run-time. Instead, DOMO consults the Java EE deployment descriptor to identify such inter-component calls and models the observed behavior, independent of the container implementation. Effectively, DOMO ignores the generated, deployed code, and models the application-level semantics of inter-component calls directly, based on direct analysis of the deployment descriptor. This functionality is necessary for accurate analysis of inter-component calls in Java EE, and to our knowledge is not supported by any other static analysis implementation.

For the code above, the call graph will contain an additional edge that links the declaration of `m2` in `Bean2` to the actual (remote) implementation of `m2` in `Bean2Bean`, as shown in Figure 2.

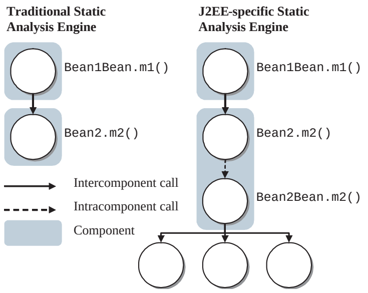

Figure 2. Traditional Static Analyzer vs. ESPE

ESPE also includes code to automatically identify application entry points, which correspond to EJB interfaces, servlets, JavaServer Pages (JSP) applications, Struts, message-driven beans, and Java EE application clients. The call graph construction resolves behavior from these various types of entry points appropriately, depending on the semantics of each type as specified by Java EE [30].

6 Experimental Results

This section summarizes the experimental results obtained by using ESPE on the following Java EE V1.4 applications: `PetStore` [29], `Bookstore` [7], `EnrollerApp` [28], `SavingsAcc` [28], `DukesBank` [28], `ITSOBank` [18], `Trade3` [12], `SPECj2002` [26], and anonymous production-level applications `A` and `B`.

The results reported in Table 1 are from running ESPE on an IBM T60P ThinkPad with an Intel T2600 Core Duo 2.16 GHz processor, 2 GB of Random Access Memory (RAM), and Microsoft Windows XP SP2 operating system. ESPE was run on a Java SE V1.5.0_07 run-time environment. For each application, Table 1 shows the application size (which does not include the libraries), the size of the call graph generated by ESPE, the time taken to perform the analysis, the total amount of memory (Java heap size) required to perform the analysis, the number of roles defined by the application, and the number of problems reported by ESPE, characterized as insufficiency, redundancy, and subversion problems. We further classify each problem as arising from role assignments to users (υ) or role assignements from principal delegation policies (π).

DukesBank, ITSOBank, and commercial applications A and B came with predefined roles, while for the other applications it was necessary for a system administrator to define the security policy based on the introspection performed on the applications by Sun Microsystems' Deployment Tool for Java 2 Platform Enterprise Edition 1.4. In this task, the system administrator attempted to configure valid security policies, and did not intend to introduce any security flaws.

The experiments in this paper rely on call graph construction via Rapid Type Analysis (RTA) [2], supporting analysis of large Java EE applications in relatively short time.

The results shows that the analysis discovered a number of policy problems in each application, spanning the three types of policy flaws we have identified. Most of the flaws stem from user role assignments as opposed to principal delegation policies. Notably, the analysis discovered thirty-eight flaws in the four applications which came configured with RBAC policies, including ten flaws in the two production codes A and B from IBM customers.

As explained earlier, the redundancy analysis is complete, so all redundancy reports identify actual violations of the Principle of Least Privilege. The sufficiency and subversion analyses are sound but not complete, and so subject to false positives. We examined each of the insufficiency and subversion problems reported by ESPE by hand, in order to identify false positives.

Somewhat surprisingly, none of the reported problems appear to be false positives, despite overapproximations in the relatively imprecise RTA call graph. We believe that the calling patterns in these Java EE programs *that affect RBAC analysis* are predominantly monomorphic, and thus amenable to context-insensitive call graph analysis. In practice, most enterprise beans map directly from the structure of an underlying relational database, and so do not utilize inheritance or linked structures. Furthermore, applications rarely store or manipulate EJB instances with complex heap data structures. Although the underlying container utilizes complex libraries and data structures, the DOMO analyzer truncates paths into the container, so container code does not pollute the application-level call graph. Furthermore, interactions with Java standard libraries are usually uninteresting for role analysis, since library methods are not restricted with roles.

We now discuss the problems identified in the four codes which came with pre-configured RBAC policies. We first discuss the problems in the two production commercial applications in more detail, since these represent real problems from the field.

Commercial application A contained three insufficiency errors that were not detected during testing. The architecture of application A assumed that four of its archive files be installed on an Internet environment, while the remaining three archive files be installed on an intranet environment for use by customer service representatives. During testing, paths of execution initiated in the intranet and involving components of both environments were never explored, and three role requirements were not identified, leading to three insufficiency problems. Those paths, which were valid, were however identified by ESPE, and the security policy was corrected before any user could experience a run-time authorization failure.

Production-level application B contained seven security problems, one of which was due to the user role assignment function's being insufficient. This vulnerability was interesting because it was also difficult to discover without an automated tool. In an RBAC system, certain resources can be marked as *inaccessible*, meaning that no user can access them regardless of the roles the user has been granted. One of the methods in B was marked as inaccessible because it was supposed to be executed only for debugging purposes, when security is disabled. ESPE detected a path that exposed that inaccessible method to indirect invocation by other programs. If that path had inadvertently been exploited at run time, it would have caused an authorization failure. That problem was not detected during testing.

The ITSOBank program exposes a redundant policy. For example, when method onMessage in IncomingTransferBean invokes getBalance and setBalance on BranchAccountLocal, its component sets the principal delegation to mdbuser, but this role is not necessary to execute any of the methods transitively reached from that execution point, which makes π redundant. Similarly, entry point Consultation.getBranchBalance has been restricted with role formula accountant $\vee$ manager $\vee$ clerk $\vee$ consultant. A system administrator granting a user the right to execute this entry point may either choose any of these roles, or all of them. ESPE has detected that roles manager $\wedge$ clerk $\wedge$ consultant are transitively necessary to execute the application starting at that entry point, but role accountant is not.

Both DukesBank and ITSOBank present cases of subversive policies. For example, in DukesBank, the Dispatcher component sets the principal-delegation policy to bankAdmin, but this role would not be sufficient to execute the intra-component method invocations restricted with bankCustomer if authorization

Application	Size	Call Graph		Time	Mem.	Roles	Problems					
	(KB)	Nodes	Edges	(sec.)	(MB)		Insufficiency		Redundancy		Subversion	
							υ	π	υ	π	υ	π
PetStore	1,282	6,465	23,360	35	117	3	8	0	0	5	0	0
Bookstore	359	16,269	86,448	62	162	3	2	2	0	14	0	0
EnrollerApp	15	2,212	10,060	12	220	3	1	0	0	0	0	0
SavingsAcc	10	2,164	9,799	12	227	4	1	0	0	0	0	0
DukesBank	149	3,452	9,322	23	212	2	1	0	0	1	0	3
ITSOBank	1,388	11,448	42,220	59	150	7	10	0	6	3	0	4
Trade3	2,414	5,634	20,655	29	114	2	21	0	0	0	1	0
SPECj2002	3,608	5,536	20,614	52	150	2	31	0	0	13	2	3
A	2,580	618	1,007	11	239	4	3	0	0	0	0	0
B	12,889	15,527	78,434	19	241	6	1	2	2	2	0	0

Table 1. Empirical Results of ESPE Analysis

were enforced regardless of component boundaries. In ITSOBank, the component IncomingTransferBean enforces principal delegation by assigning the user the identify of role mdbuser. However, this principal delegation policy would not be sufficient for the user to pass the subsequent internal authorization test for manager ∨ clerk ∨ consultant.

Overall, we conclude that ESPE is effective in identifying flaws in security policies. The tool has been applied by various groups in IBM working with customer enterprise applications.

7 Related Work

Ferraiolo and Kuhn proposed RBAC in 1992 [9]. Work on building and analyzing models and implementations for RBAC has concentrated on complex architectures [24]. Schaad, et al. [25] used the Alloy specification language [13] for modeling RBAC96, and the Alloy Constraint Analyzer (Alcoa) [14] to check desirable properties such as separation of duties.

eXtensible Markup Language (XML) documents are often used by Web applications. Several mechanisms and frameworks for specification and enforcement of RBAC policies for XML documents have been proposed [5, 15]. Such mechanisms are flexible in the sense that they prohibit or allow access to specific individual elements in XML documents. Recently, Murata, et al. [16] proposed a static analysis approach based on finite state automata that alleviates the burden of enforcement of such specifications at run time. A positive side effect of this work is faster execution of queries over XML documents.

In the area of Web applications, a number of testing and static analysis techniques have been studied, but they have concentrated primarily on the problem of control and information flow between static and dynamic resources utilized by Web applications. For example, Ricca, et al. [22] introduced a Unified Modeling Language (UML) model for Web applications that is useful for structural testing. However, this model concentrates on links between Web pages and interactive features of Web applications, and does not provide support for distributed object components. The purpose of the work of Clarke, et al. [4] is to enforce confinement of EJB objects. An EJB object's confinement can be breached when a direct reference to the EJB object is returned to a client, thereby allowing a client to invoke security-sensitive methods bypassing any RBAC restriction. They proposed coding guidelines that, if observed, prevent confinement breaches. Additionally, they described a straightforward static analysis algorithm that checks for violations of those guidelines in EJB programs. Naumovich and Centonze [17] identified the need for mapping a method-based RBAC policy to the underlying data. Subsequently, Centonze, et al. [3] proved necessary and sufficient conditions under which a method-based RBAC policy admits an equivalent data-based RBAC policy, and proposed a static analysis model to determine whether an RBAC policy is inconsistent (for example, two EJB methods accessing the same data in the same mode are not restricted with the same roles).

In addition to RBAC, Java offers a low-level access control mechanism to protect static resources, such as the file system, network, and operating system. Both static and dynamic analysis techniques are employed in modeling security and authorization. Much of this work has been applied to eliminate or minimize redundant authorization tests and identify the minimal security policy to execute a Java SE application without authorization failures [20].

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading,

MA, USA, Jan. 1986.

[2] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–341, San Jose, CA, USA, 1996. ACM Press. Also published in ACM SIGPLAN Notices 31(10).

[3] P. Centonze, G. Naumovich, S. J. Fink, and M. Pistoia. Role-Based Access Control Consistency Validation. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 121–132, Portland, ME, USA, 2006. ACM Press.

[4] D. Clarke, M. Richmond, and J. Noble. Saving the World from Bad Beans: Deployment-Time Confinement Checking. In *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 374–387, Anaheim, CA, USA, 2003. ACM Press.

[5] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A Fine-grained Access Control System for XML Documents. *ACM Transactions on Information Systems Security*, 5(2):169–202, 2002.

[6] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.

[7] H. M. Deitel, P. J. Deitel, and S. E. Santry. *Advanced Java 2 Platform: How to Program*. Prentice Hall, Upper Saddle River, NJ, USA, September 2001.

[8] Eclipse Project, `http://www.eclipse.org`.

[9] D. F. Ferraiolo and D. R. Kuhn. Role-Based Access Controls. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pages 554–563, Baltimore, MD, USA, October 1992.

[10] A. Freeman and A. Jones. *Programming .NET Security*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, June 2003.

[11] D. Grove and C. Chambers. A Framework for Call Graph Construction Algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, November 2001.

[12] IBM Corporation, Trade3 Benchmark, `http://www.ibm.com/software/`.

[13] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[14] D. Jackson, I. Schechter, and H. Shlyahter. Alcoa: The Alloy Constraint Analyzer. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 730–733, Limerick, Ireland, 2000. ACM Press.

[15] M. Kudo and S. Hada. XML Document Security Based on Provisional Authorization. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 87–96, Athens, Greece, Nov. 2000. ACM Press.

[16] M. Murata, A. Tozawa, M. Kudo, and S. Hada. XML Access Control Using Static Analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 73–84, Washington, DC, USA, Oct. 2003. ACM Press.

[17] G. Naumovich and P. Centonze. Static Analysis of Role-Based Access Control in J2EE Applications. *SIGSOFT Software Engineering Notes*, 29(5):1–10, Sept. 2004.

[18] J. Picon, P. Genchi, M. Sahu, M. Weiss, and A. Dessureault. *Enterprise JavaBeans Development Using VisualAge for Java*. IBM Redbooks. IBM Corporation, International Technical Support Organization, San Jose, CA, USA, June 1999.

[19] M. Pistoia, S. J. Fink, R. J. Flynn, and E. Yahav. When Role Models Have Flaws: Static Validation of Enterprise Security Policies. Technical report, IBM Corporation, Thomas J. Watson Research Center, Yorktown Heights, NY, USA, Sept. 2006.

[20] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, Glasgow, Scotland, UK, July 2005. Springer-Verlag.

[21] M. Pistoia, N. Nagaratnam, L. Koved, and A. Nadalin. *Enterprise Java Security*. Addison-Wesley, Reading, MA, USA, February 2004.

[22] F. Ricca and P. Tonella. Analysis and Testing of Web Applications. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, Toronto, ON, Canada, 2001. IEEE Computer Society.

[23] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, Sept. 1975.

[24] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, Feb. 1996.

[25] A. Schaad and J. D. Moffett. A Lightweight Approach to Specification and Analysis of Role-Based Access Control Extensions. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies*, pages 13–22, Monterey, CA, USA, 2002. ACM Press.

[26] Standard Performance Evaluation Corporation Java Business Benchmark 2000 (SPECjbb2000), `http://www.spec.org`.

[27] Sun Microsystems, Enterprise JavaBeans$^{\text{TM}}$ Specification, `http://java.sun.com/products/ejb/`.

[28] Sun Microsystems, J2EE 1.4 Tutorial, `http://java.sun.com/j2ee/1.4/download.html#tutorial/`.

[29] Sun Microsystems, Java PetStore, `http://java.sun.com/developer/releases/petstore/`.

[30] Sun Microsystems, Java$^{\text{TM}}$ Platform, Enterprise Edition Specification, `http://java.sun.com/j2ee`.

A Semilattice Isomorphism

This section shows that the set-based dataflow analysis described in Section 4.2 is isomorphic to an analysis formulated directly using predicate logic on role requirements.

Let R be the finite set of role predicates for p and $\text{MCNF}(R) \subseteq \mathcal{B}(R)$ be the set of propositional logic statements in *monotone* (no negations) conjunctive normal form over R. It is easy to see that $\mathcal{L}_{\mathcal{B}} := \left(\frac{\text{MCNF}(R)}{\Leftrightarrow}, \wedge\right)$ is a join semilattice, with an induced partial order corresponding to implication, $\Rightarrow$.

Consider now the subset $\mathcal{D}(R) := \{\{R_i\}_{i=1}^k \in \mathcal{P}(\mathcal{P}(R)) : \forall j \in [1,k], j \neq i \Rightarrow R_i \not\subseteq R_j\}$. It is easy to prove that $\forall X \in \mathcal{P}(\mathcal{P}(R))$, X can be canonically reduced to its subset $\chi(X) \in \mathcal{D}(R)$, called the *canonical form* of X, obtained by eliminating from X those elements that are supersets of other elements. This defines a function $\chi : \mathcal{P}(\mathcal{P}(R)) \to \mathcal{D}(R)$. The relation $\sim$ on $\mathcal{P}(\mathcal{P}(R))$ defined by $X \sim Y \overset{\text{def}}{\iff} \chi(X) = \chi(Y)$ is an equivalence relation, and $\mathcal{L}_{\mathcal{S}} := \left(\frac{\mathcal{P}(\mathcal{P}(R))}{\sim}, \cup\right)$ is a join semilattice, with induced partial order $\supseteq$.

We will prove $\mathcal{L}_{\mathcal{S}}$ to be isomorphic to $\mathcal{L}_{\mathcal{B}}$ through the function $\phi : \mathcal{L}_{\mathcal{B}} \to \mathcal{L}_{\mathcal{S}}$ defined as follows, $\forall \sigma \in$ MCNF(R), with $\sigma = \bigwedge_{i=1}^k \bigvee_{j=1}^{t_i} r_{ij}$:

$$\phi(\sigma) := \{R_i\}_{i=1}^k, \text{where } R_i := \{r_{ij}\}_{j=1}^{t_i}, \forall i = 1, \ldots, k$$

To prove that ϕ is an isomorphism of semilattices, we will show that (i) ϕ is well defined, (ii) ϕ is invertible, and (iii) ϕ is a homomorphism of semilattices.

Proposition A.1. *$\forall X = \{R_i\}_{i=1}^k \in \mathcal{P}(\mathcal{P}(R))$, with $R_i = \{r_{ij}\}_{j=1}^{t_i}, \forall i = 1, \ldots, k$, let $\psi(X) := \bigwedge_{i=1}^k \bigvee_{j=1}^{t_i} r_{ij}$. Then $\psi(\phi(\sigma)) \Leftrightarrow \sigma, \forall \sigma \in$ MCNF(R).*

Proof. $\psi(\phi(\sigma))$ and σ differ at most by the order of the disjuncts and of the predicates inside each disjunct, and by the fact that in $\psi(\phi(\sigma))$ there are no duplicate disjuncts. □

An element $\sigma' \in$ MCNF(R) is said to be in *canonical form* if σ' has no duplicate disjuncts and $\phi(\sigma') \in \mathcal{D}(R)$. Proposition A.2 shows that any $\sigma \in$ MCNF(R) has an equivalent canonical form σ'.

Proposition A.2. *$\forall \sigma \in$ MCNF(R), let $\sigma' := \psi(\chi(\phi(\sigma))) \in$ MCNF(R). Then $\chi(\phi(\sigma')) = \phi(\sigma')$ and $\sigma \Leftrightarrow \sigma'$.*

Proof. Let $\sigma = \bigwedge_{i=1}^k \bigvee_{j=1}^{t_i} r_{ij}$ and $\phi(\sigma) = \{R_i\}_{i=1}^k$, where $R_i := \{r_{ij}\}_{j=1}^{t_i}, \forall i = 1, \ldots, k$. Assume $\exists q, s \in \{1, \ldots, k\} : q \neq s \wedge R_q \subseteq R_s$. Clearly, $\bigvee_{j=1}^{t_q} r_{qj} \wedge \bigvee_{j=1}^{t_s} r_{sj} \Leftrightarrow \bigvee_{j=1}^{t_q} r_{qj}$, so $\psi(\{R_i\}_{i \in 1,\ldots,k \setminus \{s\}}) \Leftrightarrow \sigma$. Eliminating from $\phi(\sigma)$ all the elements that are supersets of other elements yields $\chi(\phi(\sigma))$, which implies that σ' has the desired properties. □

Lemma A.1. *If $\sigma_1' = \bigwedge_{i=1}^k \bigvee_{j=1}^{t_i} x_{ij}, \sigma_2' = \bigwedge_{q=1}^h \bigvee_{s=1}^{l_q} y_{qs}$ are in canonical form and $\exists i \in \{1, \ldots, k\}, q \in \{1, \ldots, h\} : X_i \subset Y_q$, with $X_i = \{x_{ij}\}_{j=1}^{t_i}, Y_q = \{y_{qs}\}_{s=1}^{l_q}$, then $\sigma_1' \not\Leftrightarrow \sigma_2'$.*

Proof. Since $X_i \subset Y_q$, $\exists y_q \in Y_q \setminus X_i$, and since σ_2' is in canonical form, $\forall l \in \{1, \ldots, h\} \setminus \{q\}$, it must be $Y_l \not\subseteq Y_q$, which implies that $\exists y_l \in Y_l \setminus Y_q$, and so $y_l \notin X_i$. Thus, the following truth assignment is possible, which demonstrates $\sigma_1' \not\Leftrightarrow \sigma_2'$:

- $y_l :=$ `true`$, \forall l = 1, \ldots, h$, which makes σ_2' `true`
- $x :=$ `false`$, \forall x \in X_i$, which makes σ_1' `false` □

Theorem A.1. *$\forall \sigma_1, \sigma_2 \in$ MCNF$(R), \sigma_1 \Leftrightarrow \sigma_2$ if and only if $\phi(\sigma_1) \sim \phi(\sigma_2)$.*

Proof. If $\chi(\phi(\sigma_1)) = \chi(\phi(\sigma_2))$, then $\psi(\chi(\phi(\sigma_1))) = \psi(\chi(\phi(\sigma_2)))$, and by Proposition A.2, $\sigma_1 \Leftrightarrow \sigma_2$.

Now, assume $\sigma_1 \Leftrightarrow \sigma_2$. By Proposition A.2, $\sigma_1' := \psi(\chi(\phi(\sigma_1))), \sigma_2' := \psi(\chi(\phi(\sigma_2')))$ are in canonical form, and $\sigma_2' \Leftrightarrow \sigma_2 \Leftrightarrow \sigma_1 \Leftrightarrow \sigma_1'$. Let $\chi(\phi(\sigma_1)) = \{X_i\}_{i=1}^k, \chi(\phi(\sigma_2)) = \{Y_j\}_{j=1}^h$, and assume by contradiction that $\exists i \in \{1, \ldots, k\} : X_i \neq Y_j, \forall j = 1, \ldots, h$. By Lemma A.1, $Y_j \not\subseteq X_i, \forall j = 1, \ldots, h$, and so $\forall j = 1, \ldots, h, \exists y_j \in Y_j \setminus X_i$, making the following truth assignment possible:

- $y_j :=$ `true`$, \forall j = 1, 2, \ldots h$, which makes σ_2' `true`
- $x :=$ `false`$, \forall x \in X_i$, which makes σ_1' `false`

This implies $\sigma_1' \not\Leftrightarrow \sigma_2'$, a contradiction. □

Thus, ϕ is well defined, with inverse ψ. From Theorem A.1, it follows immediately that ϕ also preserves the structures of $\mathcal{L}_{\mathcal{B}}$ and $\mathcal{L}_{\mathcal{S}}$ with respect to the binary operators $\wedge$ and $\cup$. Thus ϕ is an isomorphism of join semilattices, which maps $\Rightarrow$ in $\mathcal{L}_{\mathcal{B}}$ to $\supseteq$ in $\mathcal{L}_{\mathcal{S}}$. This allows us to faithfully cast the Role Requirement Analysis into a dataflow problem over $\mathcal{L}_{\mathcal{S}}$.