Incremental Information Flow Analysis of Role Based Access Control

Mikhail I. Gofman, Ruiqi Luo, Jian He, Yingbin Zhang, and Ping Yang
Dept. of Computer Science, State University of New York at Binghamton, Binghamton, NY, 13902

Abstract—*Role-Based Access Control (RBAC) has been widely used for expressing access control policies. Although RBAC provides flexible mechanisms to control the access to information, it does not control how the information propagates after it is obtained. Formally analyzing information flows resulting from an RBAC policy helps administrators understand the policy and detect potential flaws in the policy. Further, RBAC policies tend to evolve incrementally over time and it would be inefficient to perform analysis from scratch upon every change to the policies. Incremental analysis is useful in situations where small changes to the policy lead to small or no changes to the analysis result. In this paper, we present the first algorithms for incrementally analyzing information flows whenever a change is made to an RBAC policy. The performance results show that our incremental algorithms significantly outperform our non-incremental algorithm in terms of execution time while requiring only moderately larger disk space.*

Keywords: information flow, role-based access control, incremental analysis

1. Introduction

Role-based access control (RBAC) [6] has been widely used for expressing access control policies. The central notion of RBAC is that users are assigned appropriate roles, and roles are assigned appropriate permissions. A role is typically a job function or a position in an organization such as doctor, nurse, or patient. Expressing access control policies using roles eases administration and reduces redundancies in the policy.

Although RBAC provides a flexible mechanism to control the release of information, it does not control how the information is propagated after it is obtained. For example, if a user has permission to read from an object O_1 and to write to an object O_2, then this user can write the content of O_1 to O_2. As a consequence, a user who has permission to access O_2 but not O_1 will be able to read the content of O_1 through O_2. Formally analyzing information flows allowed by an RBAC policy helps security administrators understand the policy better and detect potential flaws in the policy. Due to their sheer size and complexity, the full implications of RBAC policies in large organizations can be difficult to understand by manual inspection alone. To address this problem, Osborn [5] presented an algorithm for constructing an *information flow graph* from an RBAC policy. Each node in the information flow graph is an object. There is an edge $O_1 \rightarrow O_2$ in the graph if information can flow directly from object O_1 to object O_2, i.e., if there exists a user who has the permission to read from O_1 and write to O_2. In this paper, we optimize this algorithm to reduce the time and space consumed for constructing the information flow graph. We also support queries such as "can information flow, directly or transitively, from object O_1 to object O_2?" and provide the corresponding evidence.

Further, RBAC policies tend to evolve incrementally over time, due to changes performed through either Administrative Role-Based Access Control (ARBAC) or other Administrative models. It would be inefficient for the analysis algorithm to reconstruct the graph from scratch if changes to an RBAC policy result in small or no changes to the information flow graph. In this paper, we present algorithms for incrementally updating the information flow graph upon changes to policies by reusing the results obtained from the previous analysis. All of our incremental algorithms have the same or better worst-case complexity than our non-incremental algorithm. We have also developed a university RBAC policy and an algorithm for randomly generating a relation based on the university RBAC policy and used them to compare our incremental algorithms against our non-incremental algorithm. The performance results show that our incremental algorithms significantly outperform our non-incremental algorithm in terms of time. Our incremental algorithms also require only moderately larger disk space, which is smaller than the size of the policy.

Organization. The rest of the paper is organized as follows. Section 2 provides a brief overview of RBAC. Section 3 presents the detailed optimizations to the algorithm in [5]. The incremental analysis algorithms are given in Section 4. Information flow queries and information flow analysis for parameterized RBAC are considered in Section 5. Section 6 compares the performance of our incremental algorithms against our non-incremental algorithm.

2. Background: Role Based Access Control

In role-based access control, users are assigned roles and roles are assigned permissions. Let UA be a set of user-role relations, PA be a set of permission-role relations, and

DSD be a set of dynamic separation of duty constraints. The user-role relation $(U, R) \in UA$ specifies that the user U is a member of role R. Permission-role relation $(R, O, P) \in PA$ specifies that all members of a role R can perform the operation P (e.g. read, write) on an object O. The dynamic separation of duty constraint $(R_1, R_2) \in DSD$ specifies that a user cannot invoke both R_1 and R_2 simultaneously in one session. Role hierarchy in RBAC defines a partial order on the set of roles. A role hierarchy relation $R_1 \succeq R_2$ specifies that role R_1 is senior to role R_2, i.e., every permission assigned to R_2 is available to members of R_1. We use $R_1 \succ R_2$ to specify that $R_1 \succeq R_2$ and $R_1 \neq R_2$.

3. Information Flow Analysis of RBAC

Algorithm 1 Information Flow Analysis Algorithm

```
Procedure $info\_graph()$
trans_pa_rh()
trans_dsd_rh()
for all $(R, O_1, r), (R, O_2, w) \in PA$ such that $O_1 \neq O_2$ do
    if $R$ appears in UA then
        $add\_edge((R, O_1) \rightarrow (R, O_2))$
    end if
end for
for all $U \in User$ do
    for all $(U, R_1), (U, R_2) \in UA$ do
        if $R_1 \neq R_2$ and $(R_1, R_2) \notin DSD$ then
            for all $(R_1, O_1, r), (R_2, O_2, w) \in PA$ where $O_1 \neq O_2$ do
                $add\_edge((R_1, O_1) \rightarrow (R_2, O_2))$
            end for
        end if
    end for
end for
for all $O \in Obj$ do
    for all $(R_1, O, w), (R_2, O, r) \in PA$ where $R_1 \neq R_2$ do
        if both $R_1$ and $R_2$ appear in UA then
            $add\_edge((R_1, O) \rightarrow (R_2, O))$
        end if
    end for
end for

Procedure $trans\_pa\_rh()$
for all $R_1 \in Role$ do
    $junior(R_1) = |\{R | R_1 \succ R\}|$
end for
$W = \{R_1 | junior(R_1) = 0)\}$
while $W \neq \emptyset$ do
    remove $R_1$ from $W$
    for all $R_2 \succeq R_1$ do
        for all $(R_1, O, p) \in PA$ do
            add_perm $(R_2, O, p)$
        end for
        if $(--junior(R_2) == 0)$ then
            $W = W \cup \{R_2\}$
        end if
    end for
end while

Procedure $trans\_dsd\_rh()$
$W = \{(R_1, R_2) | (R_1, R_2) \in DSD\}$
while $W \neq \emptyset$ do
    remove $(W_1, W_2)$ from $W$
    for all $(SW_1 \succeq W_1)$ do
        for all $(SW_2 \succeq W_2)$ do
            if $(SW_1, SW_2)$ is not already in DSD then
                add $(SW_1, SW_2)$ to DSD
                add $(SW_1, SW_2)$ to $W$
            end if
        end for
    end for
end while
```

In this section, we first summarize the information flow analysis algorithm presented in [5], and then present optimizations to improve its performance.

Information flow analysis algorithm in [5]. Osborn [5] proposed an algorithm for generating an information flow graph from an RBAC policy. For simplicity, the author assumes the only operations on objects are read and write; other permissions can be represented as read and write permissions. The algorithm consists of two stages. In the first stage, the algorithm generates a cyclic information flow graph from the RBAC policy. There is an edge $(R_1, O_1, P_1) \rightarrow (R_2, O_2, P_2)$ in the graph if one of the following three conditions is satisfied: (1) $P_1 = r$, $P_2 = w$, a user is assigned both roles R_1 and R_2, $(R_1, R_2) \notin DSD$, R_1 can read from O_1, and R_2 can write to O_2; (2) $P_1 = r$, $P_2 = w$, $R_1 = R_2$, R_1 appears in UA, and R_1 can read from O_1 and write to O_2; or (3) $O_1 = O_2$, both R_1 and R_2 appear in UA, R_1 has privilege P_1 on O_1, and R_2 has privilege P_2 on O_1. In the second stage, the algorithm merges nodes in every cycle of the graph, which produces an acyclic graph.

Optimizations. We optimize the algorithm in [5] to reduce the size of the cyclic graph generated in the first stage as well as the overhead of the second stage for detecting and eliminating cycles. First, our algorithm avoids generating edges that do not capture actual flows of information, but are used in [5] to construct cycles. Such edges are of the form $(R_1, O, r) \rightarrow (R_2, O, r)$, $(R_1, O, w) \rightarrow (R_2, O, w)$, and $(R_1, O, r) \rightarrow (R_2, O, w)$. Second, the r/w privilege is removed from the graph as it can be inferred by examining the objects in the edge: given an edge $(R_1, O_1) \rightarrow (R_2, O_2)$, if $O_1 = O_2$, the edge represents $(R_1, O_1, w) \rightarrow (R_2, O_2, r)$; otherwise, the edge represents $(R_1, O_1, r) \rightarrow (R_2, O_2, w)$. Such a simplification reduces the size of the cyclic graph. An acyclic graph can be generated by removing roles from the nodes of the graph and removing self-edges, and then merging nodes in each cycle to one node.

Our algorithm for constructing the cyclic information flow graph is given in Algorithm 1. Procedure $trans\_pa\_rh$ translates the hierarchical permission-role relations into non-hierarchical relations: every role inherits all permissions of its junior roles. The translation process is performed bottom-up (i.e., from junior roles to senior roles) and guarantees that each role in the role hierarchy will be processed at most once. Procedure $trans\_dsd\_rh$ converts hierarchical DSD constraints to non-hierarchical DSD constraints. Procedure $add\_perm(perm)$ checks if a permission $perm$ is in PA, and if not, adds $perm$ to PA. Procedure $add\_edge(e)$ checks if an edge e already exists in the information flow graph, and if not, adds e to the graph. Due to space constraints, we omit the details of procedures add_edge and add_perm.

In our implementation, we use adjacency-list to store the information flow graph. The worst-case complexity of Algorithm 1 is $O(|Role|^5 + |User||Role|^3|Obj|^3)$ where

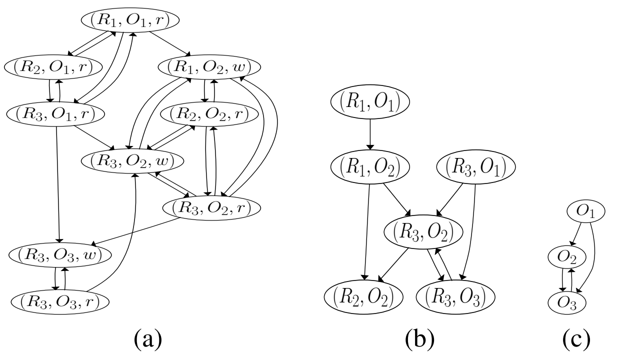


Fig. 1: (a) The information flow graph generated using the algorithm in [5]. (b) The information flow graph generated using Algorithm 1. (c) The information flow graph containing only objects.

$User$, $Role$, and Obj are sets of users, roles, and objects, respectively.

Example 1. *Consider the following RBAC policy P.*
UA $=\{(U_1,R_3),\ (U_2,R_3),\ (U_3,R_2),\ (U_4,R_1),\ (U_5,R_1)\}$
PA $= \{(R_1,O_1,r),(R_1,O_2,w),(R_2,O_1,r),(R_2,O_2,r),$ $(R_3,O_3,r),(R_3,O_3,w)\}$
DSD $=\emptyset$, ***RH*** $=\{R_3 \succ R_1,\ R_3 \succ R_2\}$

Figure 1(a) gives the cyclic information flow graph generated from policy P using the algorithm in [5], which contains 9 nodes and 25 edges. The information flow graph generated from Algorithm 1 is given in Figure 1(b), which contains 6 nodes and 8 edges. By removing roles from nodes in Figure 1(b) and deleting self edges, we obtain a graph containing only objects (Figure 1(c)). The acyclic graph generated using the algorithm in [5] contains one edge $\{O_2,O_3\} \rightarrow \{O_1\}$, which can be generated from Figure 1(c) by merging nodes containing objects O_2 and O_3.

Discussion. The algorithm in [5] considers the information flows resulting from every individual user in the RBAC policy. We observe that users assigned the same set of roles result in identical information flows. Thus, it is sufficient to consider only users with distinct sets of roles, which is usually significantly smaller than the total number of users. Users assigned the same set of roles are said to be in the same *equivalence class*. For example, in a university RBAC policy we developed, there are 15988 users and 33 equivalence classes. However, computing the set of equivalence classes has the worst-case complexity of $O(|User|^2|Role|^2)$, which increases the worst-case complexity of Algorithm 1. The performance results show that using equivalence classes does not improve the performance of Algorithm 1 on the university RBAC policy (Section 6).

4. Incremental Information Flow Analysis

In this section, we present algorithms for incrementally updating the information flow graph in response to changes to the RBAC policy. Our incremental algorithms reuse the results of the previous analysis, including the information flow graph, the non-hierarchical permission-role relations, and the non-hierarchical DSD constraints. All of our incremental algorithms have the same or better complexity results than Algorithm 1 (the non-incremental algorithm).

Revising Algorithm 1 to support incremental analysis. To support incremental analysis, we associate each permission-role relation, DSD constraint, and edge in the information flow graph with a counter. The initial values of all counters are 0. Every time a permission-role relation, a DSD constraint, or an edge is added, the corresponding counter is increased by 1. Note that the counter of a permission-role relation may be smaller than the total number of derivations of the relation. For example, the counter of a permission-role relation (R_1,O,p) is defined as: $counter(PA(R_1,O,p)) = ((R_1,O,p) \in PA?1:0) + |\{R|R_1 \succeq R \wedge counter(PA(R,O,p)) > 0\}|$. Assume that $RH=\{R_3 \succ R_1, R_3 \succ R_2, R_2 \succ R_4\}$ and $PA=\{(R_1,O,p),(R_2,O,p),(R_3,O,p),(R_4,O,p)\}$. Then $counter(PA(R_4,O,p)) = counter(PA(R_1,O,p)) = 1$, $counter(PA(R_2,O,p))=2$, and $counter(PA(R_3,O,p)) = 3$, but the number of derivations of (R_3,O,p) is 4. The advantage of such a representation is that, if (R_4,O,p) is deleted from the policy, we simply decrement the counter of (R_2,O,p) by 1 and the counters of other relations remain the same. However, if the counter records the total number of derivations, we will also need to decrement the counter of (R_3,O,p). Similarly, the counter of an edge of the information flow graph is also smaller than the total number of derivations of this edge.

Incremental analysis algorithms. Due to space constraints, we provide the pseudo-codes for only some of the incremental algorithms.

Add a user-role relation. When a new user-role relation $(U,R) \in UA$ is added to the RBAC policy, the incremental algorithm adds information flows resulting from R and all other roles assigned to U, to the graph. The corresponding algorithm is given in Algorithm 2 and its worst-case complexity is $O(|User||Role|^2|Obj|^3)$.

Delete a user-role relation. When a user role relation $(U,R) \in UA$ is removed from the RBAC policy, the incremental algorithm checks if R has been assigned to any user other than U. If so, the counters of all edges of the form $(R,O) \rightarrow (R_1,O_1)$ and $(R_1,O_1) \rightarrow (R,O)$, where $R \neq R_1$ and $O \neq O_1$, are decreased by 1 and all edges whose counters reach 0 are deleted. Otherwise, the algorithm deletes all edges containing role R. The worst-case complexity of this algorithm is $O(|User||Role| + |Role|^2|Obj|^2)$.

Add a permission-role relation. When a permission-role relation $(R,O,r) \in PA$ is added to an RBAC policy, all

Algorithm 2 Incremental Algorithm: add_ua(U, R)

```
procedure $add\_ua(U, R)$
for all $(U, R_1) \in UA$ where $R_1 \neq R$ and $(R_1, R) \notin DSD$ do
    for all $(R, O_1, r), (R_1, O_2, w) \in PA$ where $O_1 \neq O_2$ do
        $add\_edge((R, O_1) \rightarrow (R_1, O_2))$
    end for
    for all $(R, O_1, w), (R_1, O_2, r) \in PA$ where $O_1 \neq O_2$ do
        $add\_edge((R_1, O_2) \rightarrow (R, O_1))$
    end for
end for
if $R$ has not been assigned to other users then
    for all $(R, O_1, r), (R, O_2, w) \in PA$ where $O_1 \neq O_2$ do
        $add\_edge((R, O_1) \rightarrow (R, O_2))$
    end for
    for all $O \in Obj$ do
        for all $(R, O, r), (R_1, O, w) \in PA$ where $R_1 \neq R$ do
            if there exists $(U_1, R_1) \in UA$ then
                $add\_edge((R_1, O) \rightarrow (R, O))$
            end if
        end for
        for all $(R, O, w), (R_1, O, r) \in PA$ where $R_1 \neq R$ do
            if there exists $(U_1, R_1) \in UA$ then
                $add\_edge((R, O) \rightarrow (R_1, O))$
            end if
        end for
    end for
end if
```

roles that are senior to R inherit permission (O, r) from R. The newly generated permission-role relations may result in new edges in the information flow graph. The corresponding algorithm is given in Algorithm 3 and its worst-case complexity is $O(|User||Role|^3|Obj|^2)$. Adding $(R, O, w) \in PA$ is handled similarly.

Delete a permission-role relation. When a permission-role relation $(R, O, r) \in PA$ is deleted from an RBAC policy, the counter of $(R, O, r) \in PA$ is decreased by 1. If the counter does not become 0, the algorithm terminates. Otherwise, all edges of the form $(R, O) \rightarrow (R_1, O_1)$ where $O \neq O_1$ and $(R_1, O) \rightarrow (R, O)$ are deleted. The algorithm repeats the above procedure for all roles senior to R until no more permission-role relations can be deleted. Deletion of permission-role relations $(R, O, w) \in PA$ is handled similarly. The worst-case complexity of this algorithm is $O(|Role|^2|Obj|)$.

Add a DSD constraint. The DSD constraint $(R_1, R_2) \in DSD$ specifies that no user is allowed to activate roles R_1 and R_2 simultaneously in one session. When a new DSD constraint (R_1, R_2) is added, we also add (SR_1, SR_2), where $SR_1 \succeq R_1$ and $SR_2 \succeq R_2$, to the DSD constraints. As a result, all edges of the form $(SR_1, O_1) \rightarrow (SR_2, O_2)$ and $(SR_2, O_1) \rightarrow (SR_1, O_2)$ are deleted from the information flow graph. The corresponding algorithm is given in Algorithm 4 and its worst case complexity is $O(|Role|^5 + |Role|^3|Obj|^2)$.

Note that adding a new DSD constraint may result in conflicts in an RBAC policy. For example, assume that $R_3 \succ R_1$ and $R_3 \succ R_2$. Adding $(R_1, R_2) \in DSD$ to DSD

Algorithm 3 Incremental Algorithm: $add\_pa(R, O, r)$

```
procedure $add\_pa(R, O, r)$
if $(R, O, r) \in PA$ then
    $counter(PA(R, O, r))$++
else
    add $(R, O, r)$ to $PA$
    $counter(PA(R, O, r)) = 1$
    for all $(U, R), (U, R_1) \in UA$ where $R \neq R_1$ do
        if $(R_1, R) \notin DSD$ then
            for all $(R_1, O_1, w) \in PA, O_1 \neq O$ do
                $add\_edge((R, O) \rightarrow (R_1, O_1))$
            end for
        end if
    end for
    if $R$ appears in UA then
        for all $(R, O_1, w) \in PA$ where $O_1 \neq O$ do
            $add\_edge((R, O) \rightarrow (R, O_1))$
        end for
        for all $(R_1, O, w) \in PA$ where $R_1 \neq R$ do
            if $R_1$ appears in $UA$ then
                $add\_edge((R_1, O) \rightarrow (R, O))$
            end if
        end for
    end if
    $S = \{role | role \succ R\}$
    for all $role \in S_1$ do
        $add\_pa(role, O, r)$
    end for
end if
```

results in a new DSD constraint $(R_3, R_3) \in DSD$, which can never be satisfied. Such a conflict can be easily detected before applying the incremental algorithm.

Delete a DSD constraint. Deleting a constraint $(R_1, R_2) \in DSD$ decrements the counter associated with the constraint as well as all counters derived from it through the role hierarchy. Any constraint whose counter reaches 0 is removed. Next, for each constraint $(SR_1, SR_2) \in DSD$ that is removed, the algorithm adds the information flows resulting from users who are assigned both roles SR_1 and SR_2 to the graph. The worst-case complexity of this algorithm is $O(|Role|^5 + |User||Role|^3|Obj|^3)$.

Add a role hierarchy relation. Adding a role hierarchy relation may result in new permission-role relations and new DSD constraints. When a role hierarchy relation $R_2 \succ R_1$ is added, the incremental algorithm adds permission-role relations and DSD constraints that can be derived through $R_2 \succ R_1$, and deletes all edges that are invalidated by the new DSD constraints. The algorithm is given in Algorithm 5 and its worst-case complexity is $O(|User||Role|^3|Obj|^3)$.

Delete a role hierarchy relation. Deleting a role hierarchy relation may result in deletion of permission-role relations and DSD constraints. When $R_2 \succ R_1$ is deleted, permissions of R_2 and permissions of its senior roles that are inherited from R_1 are deleted. The DSD constraints that are derived through $R_1 \succ R_2$ are also deleted. The worst-case complexity of this algorithm is $O(|User||Role|^3|Obj|^3)$.

Example 2. *Consider the RBAC policy P given in Exam-*

Algorithm 4 Incremental Algorithm: $add\_dsd(R_1, R_2)$

```
1: procedure $add\_dsd(R_1, R_2)$
2: if $(R_1, R_2) \notin DSD$ then
3:     add $(R_1, R_2)$ to $DSD$
4:     $counter(DSD(R_1, R_2)) = 1$
5: else
6:     $counter(DSD(R_1, R_2))$++
7: end if
8: $S = W = \{(R_1, R_2)\}$
9: while $W \neq \emptyset$ do
10:    remove $(W_1, W_2)$ from $W$
11:    for all $SW_1 \succeq W_1$, $SW_2 \succeq W_2$ where $(SW_1, SW_2) \neq (R_1, R_2)$ do
12:        if $(SW_1, SW_2) \in DSD$ then
13:            $counter(DSD(SW_1, SW_2))$++
14:        else
15:            add $(SW_1, SW_2)$ to $DSD$
16:            $counter(DSD(SW_1, SW_2)) = 1$
17:            $W = W \cup \{(SW_1, SW_2)\}$
18:            $S = S \cup \{(SW_1, SW_2)\}$
19:        end if
20:    end for
21: end while
22: for all $(SR_1, SR_2) \in S$ do
23:    delete all edges of the form $(SR_1, O_1) \rightarrow (SR_2, O_2)$ and $(SR_2, O_2) \rightarrow (SR_1, O_1)$ where $O_1 \neq O_2$
24: end for
```

Algorithm 5 Incremental Algorithm: Add $R_2 \succ R_1$

```
1: procedure $add\_rh(R_2, R_1)$
2: add $R_2 \succ R_1$ to $RH$
3: $S = \emptyset$
4: for all $(R_1, R_3) \in DSD$ do
5:     $W = \{R_2\}$
6:     while $W \neq \emptyset$ do
7:         Remove $W_1$ from $W$
8:         if $(W_1, R_3) \notin DSD$ then
9:             add $(W_1, R_3)$ to $DSD$
10:            $counter(DSD(W_1, R_3)) = counter(DSD(R_1, R_3))$
11:            $W = W \cup \{R | R \succ W_1\}$
12:            $S = S \cup \{(W_1, R_3)\}$
13:        else
14:            $counter(DSD(R_2, R_3))$ += $counter(DSD(R_1, R_3))$
15:        end if
16:    end while
17: end for
18: for all $(R_4, R_5) \in S$ do
19:    delete all edges of the form $(R_4, O_1) \rightarrow (R_5, O_2)$ and $(R_5, O_2) \rightarrow (R_4, O_1)$ where $O_1 \neq O_2$
20: end for
21: for all $(R_1, O, p) \in PA$ do
22:    $add\_pa(R_2, O, p)$
23: end for
```

ple 1. Figure 1(b) gives the corresponding cyclic information flow graph. Each permission is associated with a counter: $counter(PA(R_1, O_1, r)) = counter(PA(R_1, O_2, w)) = counter(PA(R_2, O_1, r)) = counter(PA(R_2, O_2, r)) = counter(PA(R_3, O_2, r)) = counter(PA(R_3, O_2, w)) = counter(PA(R_3, O_3, r)) = counter(PA(R_3, O_3, w)) = 1$, *and* $counter(PA(R_3, O_1, r)) = 2$. *When a permission-role relation* (R_3, O_1, r) *is added to P,* $counter(PA(R_3, O_1, r))$ *is increased by 1 (i.e., = 3) and the information flow graph remains the same. When a user-role relation* (U_1, R_1) *is added to P, edges* $(R_1, O_1, r) \rightarrow (R_3, O_2, w)$, $(R_1, O_1, r) \rightarrow (R_3, O_3, w)$, $(R_3, O_1, r) \rightarrow (R_1, O_2, w)$, *and* $(R_3, O_3, r) \rightarrow (R_1, O_2, w)$ *are added to the graph. When a relation* $R_3 \succ R_1$ *is deleted from the role hierarchy of P, the counter of* (R_3, O_1, r) *becomes 1 and the counter of* (R_3, O_2, w) *becomes 0. As a consequence, the edge* $(R_3, O_2) \rightarrow (R_2, O_2)$ *is deleted from the graph.*

5. Extensions

This section extends the incremental analysis algorithms to support parameterized RBAC, to query information flow properties, and to handle a sequence of additions and deletions of RBAC relations.

5.1 Information Flow Analysis of Parameterized Role Based Access Control

PRBAC (Parameterized RBAC) [10] extends the classical RBAC model with parameters to allow greater flexibility, scalability, and expressive power. Each role in PRBAC has the form $r(p_1 = x_1, \ldots, p_n = x_n)$ where p_i is a distinct parameter name and x_i is a data value or a variable symbolically representing data values. We use identifiers starting with lower-case letters to represent data values, and identifiers starting with upper-case letters to represent variables. A role is said to be *concrete* if it does not contain variables. For example, a role for students who are taking CS101 can be specified as a concrete role *Student*$(dept = cs, cid = 101)$ where $dept$ represents the department ID and cid represents the course number. Objects in PRBAC policies are parameterized in a similar manner.

Let PUA be a set of parameterized user-role relations and PPA be a set of parameterized permission-role relations. A user-role relation $(U, R) \in PUA$ in PRBAC specifies that user U is a member of a concrete role R. The permission-role relation $(R, O, P) \in PPA$ specifies that role R has permission P on an object O; R may or may not be a concrete role. Similar to RBAC, the DSD constraint in PRBAC specifies roles that cannot be invoked simultaneously in one session and the role hierarchy defines a partial order relation among roles. The information flow graph can be constructed from a PRBAC policy as follows: there is an edge $(R_1, O_1) \rightarrow (R_2, O_2)$ in the graph if (1) there exist a user U who is assigned both roles R_1 and R_2, two permission-role relations (R'_1, O'_1, r) and (R'_2, O'_2, w), and two substitutions σ_1 and σ_2 such that $R_1 = R'_1\sigma_1$, $O_1 = O'_1\sigma_1$, $R_2 = R'_2\sigma_2$, and $O_2 = O'_2\sigma_2$; or (2) there exist two permission-role relations (R'_1, O'_1, w) (R'_2, O'_1, r) and two substitutions σ_1 and σ_2 such that $R_1 = R'_1\sigma_1$, $O_1 = O'_1\sigma_1 = O'_2\sigma_2$, and $R_2 = R'_2\sigma_2$.

Example 3. *Consider the following PRBAC policy:*
PUA $= \{(u_1, Instructor(dept = cs, cid = 101)), (u_1, Chair(dept = cs))\}$
PPA $= \{(Chair(dept = D), DeptReport(dept =$

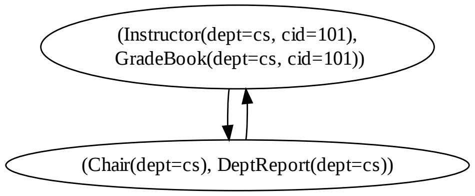

(Instructor(dept=cs, cid=101), GradeBook(dept=cs, cid=101))
(Chair(dept=cs), DeptReport(dept=cs))

Fig. 2: Information flow graph generated from the PRBAC policy in Example 3.

$D), w)$, $(Faculty(dept = D), DeptReport(dept = D), r)$, $(Instructor(dept = D, cid = C), GradeBook(dept = D, cid = C), r/w)\}$
role hierarchy $= \{Chair(dept = D) \succeq Faculty(dept = D)\}$

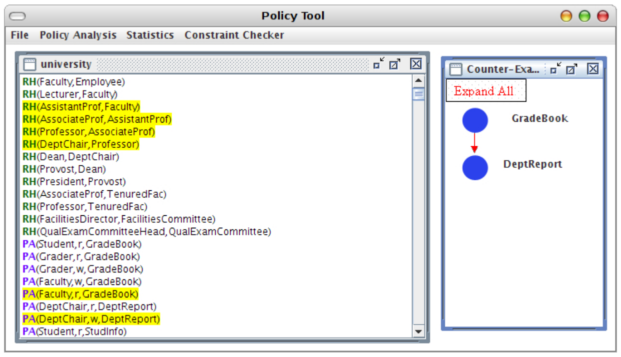
The information flow graph generated for this example is given in Figure 2.

5.2 Supporting Queries

We have developed algorithms to support the following queries about information flow properties: (1) Can information flow, directly or transitively, from an object O_1 to an object O_2? (2) From which objects the information can flow, directly or transitively, to an object O? Both queries are answered by performing depth-first search on the information flow graph generated. Figure 3 gives the screenshot for answering the query "can information flow from $GradeBook$ to $DeptReport$?" The answer to this query is yes. The right window provides diagnostic information. When clicking the edge between $GradeBook$ and $DeptReport$, the RBAC rules that result in this information flow are highlighted: information can directly flow from $GradeBook$ to $DeptReport$ because the role $DeptChair$ can read from $GradeBook$ (inherited from the role $Faculty$) and write to $DeptReport$.

5.3 Adding/Deleting a Sequence of Relations

If the administrators make multiple changes to the policy at one time, it would be sufficient to execute the incremental algorithm for every change in the order in which it was provided. Such an approach, although simple, may not always be optimal. We propose to reduce the number of changes to the information flow graph by controlling the order in which changes are processed: changes to DSD constraints are processed first, followed by changes to role-hierarchy relations. Changes to user-role and permission-role relations can be done in any order. Changes to DSD constraints are processed first because they may invalidate changes to the information flow graph caused by changes to other relations. For example, adding a user-role relation or a permission-role relation and then a DSD constraint may cause the newly added edges to be deleted. Changes to role-hierarchy relations are processed next because changes



RH(Faculty,Employee)
RH(Lecturer,Faculty)
RH(AssistantProf,Faculty)
RH(AssociateProf,AssistantProf)
RH(Professor,AssociateProf)
RH(DeptChair,Professor)
RH(Dean,DeptChair)
RH(Provost,Dean)
RH(President,Provost)
RH(AssociateProf,TenuredFac)
RH(Professor,TenuredFac)
RH(FacilitiesDirector,FacilitiesCommittee)
RH(QualExamCommitteeHead,QualExamCommittee)
PA(Student,r,GradeBook)
PA(Grader,r,GradeBook)
PA(Grader,w,GradeBook)
PA(Faculty,w,GradeBook)
PA(Faculty,r,GradeBook)
PA(DeptChair,r,DeptReport)
PA(DeptChair,w,DeptReport)
PA(Student,r,StudInfo)

Fig. 3: Information flow query for the university RBAC policy.

to role-hierarchy relations result in changes to DSD constraints.

6. Performance Results

We have developed a university RBAC policy based on the student/faculty statistics from Binghamton University. This section compares the performance of the non-incremental and incremental algorithms by executing them over a set of randomly generated RBAC relations for the university RBAC policy. All reported results were obtained on a 2.80GHz Pentium(R) D machine with 1GB RAM running Fedora Linux.

The university RBAC policy contains 33 roles, 15988 users, and 27 objects. Each data point reported is an average over 16 randomly generated RBAC relations. Due to space constraints, we only describe how we randomly generate a user-role relation to add to the university RBAC policy; other relations are generated similarly. To generate a user-role relation, we first randomly choose a *can_assign* rule from the university ARBAC policy developed in [11], which specifies the authority to assign users to roles. For example, *can_assign*$(DeptChair, Grad \wedge \neg RA, TA)$ specifies that the role $DeptChair$ has authority to assign a user who is a member of $Grad$ but not a member of RA to the role TA; TA is called the postcondition of this rule. Next, we randomly choose a user u who has not been assigned the postcondition r of the rule chosen and add (u, r) to UA.

Table 1 gives the number of nodes and edges generated, and the execution time of the following four algorithms on the university RBAC policy: the non-incremental algorithm without equivalence classes (NonInc), the non-incremental algorithm with equivalence classes (NonIncEQ), the incremental algorithm without equivalence classes (Inc), and the incremental algorithm with equivalence classes (IncEQ). Column "Operation" specifies the operations on the policy.

The performance results show that the incremental algorithms are 100-400 times faster than the non-incremental algorithm. A total of 33 equivalence classes are constructed from the university policy, which is significantly smaller than

Table 1: Performance comparison of the non-incremental algorithm and the incremental algorithms

Operation	States	Trans	Time (Sec.)			
			NonInc	NonIncEC	Inc	IncEC
add UA	142	3069	18.42	17.96	0.17	0.05
delete UA	142	3044	18.85	17.98	0.12	0.04
add PA	143	3067	18.52	18.00	0.18	0.04
delete PA	141	2918	18.77	18.33	0.12	0.04
add RH	146	3205	18.60	18.26	0.33	0.05
delete RH	121	2263	18.82	18.47	0.13	0.04
add DSD	142	2950	18.44	18.19	0.14	0.04
delete DSD	142	3044	18.85	18.23	0.17	0.04

the number of users in the policy. We observe that using equivalence classes does not improve the performance of the non-incremental algorithm due to the overhead introduced by constructing the equivalence classes from the set of users, as described in Section 3. The incremental algorithms with equivalence classes reuse the equivalence classes constructed from the previous analysis and hence avoid the above overhead. As a result, the incremental algorithms with equivalence classes are around 3 times faster than the incremental algorithms without equivalence classes. The additional amount of disk space used to store information between analysis runs is 222.9KB with equivalence classes and is 139.9KB without equivalence classes, which is less than the size of the university RBAC policy (421.6KB).

7. Related Work

Incremental computation has been applied in many different areas, including deductive databases, Logic Programming, program analysis, and model checking. The work that is most closely related to ours is Gupta et al.'s work on incrementally updating materialized views specified using Datalog [2]. While the incremental algorithms without equivalence classes can be encoded as a Datalog program, our algorithm is more efficient than directly applying their algorithm to update information flow graph due to the following reasons: (1) their algorithm was designed for general Datalog programs. Consequently, many operations performed in their algorithm are not necessary for incrementally updating the information flow graph. One of such operations involves dividing a Datalog program into its strongly connected components; and (2) the counter in their work records the total number of derivations of each Datalog relation, while the value of our counter may be less than the total number of derivations. As a result, our algorithm updates counters less frequently.

Gupta et al. [3] propose a two-phase delete-rederive algorithm (Dred) for updating materialized views in response to deletions of base relations. First, the algorithm deletes all relations in the view which depend on the deleted base relations, and then rederives deleted relations that have alternative derivations. Sokolsky et al. [9] adapt a similar approach for their incremental model checking algorithm (MCI). Saha et al. [8] also apply delete-rederive approach when facts are deleted in tabled Prolog programs. Their algorithm controls the deletion of Prolog relations based on a data structure called *support graph*. Our deletion algorithms have lower worst-case complexity than the two-phase delete-rederive algorithms when applied to information flow analysis. Also, the size of the support graph in [8] is large for large Prolog programs. Lu et al. [4] propose a Straight Delete algorithm (StDel) for updating views in constraint databases in response to deletions, which improves Dred by eliminating the rederivation phase. However, directly applying StDel to information flow analysis of RBAC would require the algorithm to store all derivations (proofs) whose size is usually large.

Some other incremental analysis algorithms do not rely on the use of counters. Conway et al. [1] present incremental analysis algorithms for C programs based on the control flow graph of the programs. Their algorithm is not directly applicable to information flow analysis of RBAC because RBAC has no control flow. Incremental computation has also been applied to data flow analysis (e.g. [7], [12]). However, they either compute less precise answers than their non-incremental counterparts or are only applicable to specific types of analysis problems.

Acknowledgement. The authors thank Weipang Liu for his contribution to the implementation.

References

[1] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *Computer Aided Verification (CAV)*, pages 449–461, 2005.

[2] A. Gupta, D. Katiyar, and I. S. Mumick. Counting solutions to the view maintenance problem. In *Workshop on Deductive Databases*, pages 185–194, 1992.

[3] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *International Conference on Management of Data*, pages 157–166, 1993.

[4] J. Lu, G. Moerkotte, J. Schu, and V. S. Subrahmanian. Efficient maintenance of materialized mediated views, 1995.

[5] S. Osborn. Information flow analysis of an RBAC system. In *ACM SACMAT*, pages 163 – 168, 2002.

[6] D. F. F. R. Sandhu and D. R. Kuhn. The nist model for role based access control: Towards a unified standard. In *ACM SACMAT*, pages 47–63, 2000.

[7] B. G. Ryder and W. A. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *International Conference on Software Engineering*, pages 442–451, 1999.

[8] D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *In International Conference on Logic Programming*, pages 392–406, 2003.

[9] O. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *International Conference on Computer Aided Verification*, pages 351–363, 1994.

[10] S. Stoller, P. Yang, M. Gofman, and C. R. Ramakrishnan. Symbolic policy analysis for parameterized administrative role based access control. In *ACM SACMAT*, 2009.

[11] S. Stoller, P. Yang, C. R. Ramakrishnan, and M. Gofman. Efficient policy analysis for administrative role based access control. In *ACM CCS*, pages 445–455, 2007.

[12] F. Vivien. Incrementalized pointer and escape analysis. In *PLDI*, pages 35–46, 2001.