Core Role-Based Access Control:
Efficient Implementations by Transformations *

Yanhong A. Liu[1]   Chen Wang[2]   Michael Gorbovitski[1]   Tom Rothamel[1]

Yongxi Cheng[2]   Yingchao Zhao[2]   Jing Zhang[2]

[1]Computer Science Department, State University of New York at Stony Brook
{liu,mickg,rothamel}@cs.sunysb.edu

[2]Computer Science Department, Tsinghua University
{wc00,cyx,yczhao,mitjj00}@mails.tsinghua.edu.cn

Abstract

This paper describes a transformational method applied to the core component of role-based access control (RBAC), to derive efficient implementations from a specification based on the ANSI standard for RBAC. The method is based on the idea of incrementally maintaining the result of expensive set operations, where a new method is described and used for systematically deriving incrementalization rules. We calculate precise complexities for three variants of efficient implementations as well as for a straightforward implementation based on the specification. We describe successful prototypes and experiments for the efficient implementations and for automatically generating efficient implementations from straightforward implementations.

Categories and Subject Descriptors   D.3.4 [*Programming Languages*]: Processors—code generation, optimization; D.4.6 [*Operating Systems*]: Security and Protection—access controls; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program transformation

General Terms   Design, Languages, Performance, Security

Keywords   access control, complexity guarantees, incrementalization, optimization, transformation

1.   Introduction

Role-based access control (RBAC) is a framework for controlling user access to resources based on roles. It can significantly reduce the cost of security policy administration and is increasingly widely used in large organizations. The ANSI standard for RBAC [1] was approved in 2004 after several rounds of public review [20, 11, 5],

building on much research during the preceding decade and earlier (e.g., [4, 8, 12]). Despite much research on RBAC, it is nontrivial to develop efficient implementations, and it is even harder to develop efficient implementations with precise complexity guarantees.

Program transformations have the promise of generating efficient implementations from high-level specifications. However, existing transformations either can not achieve asymptotic complexity improvements, or can not be fully automatically applied to real-world examples and provide complexity guarantees. Asymptotic improvements are essential to go from clear specifications to efficient implementations [14], and fully automatic transformations that provide complexity guarantees for the generated implementations are essential for real-world applications.

This paper describes a transformational method applied to the core component of RBAC, to derive efficient implementations from a specification based on the ANSI standard for RBAC. Core RBAC defines core functionalities on permissions, users, sessions, roles, and a number of relations among these sets, while the rest of RBAC adds a hierarchical relation over roles, in hierarchical RBAC, and restricts the number of roles of a user and of a session, in constrained RBAC.

The transformational method is based on the old idea of incrementally maintaining the result of expensive set operations [19], but no previous work addresses the challenge of systematically deriving incrementalization rules, which is extremely important for handling a variety of different kinds of complex set queries such as those in RBAC. We illustrate our method on core RBAC as a non-trivial case study, and calculate precise complexities for three variants of efficient implementations as well as for a straightforward implementation based on the specification, for comparisons. The method described is general and can incrementalize expensive set expressions in any application, including other variants and extensions of core RBAC.

We have developed prototypes for both straightforward and incrementalized implementations, as well as a system for automatically applying incrementalization rules to generate incrementalized implementations, and performed a large number of experiments to test for correctness and to confirm the calculated complexities. Our method of incrementalization supports the separation of "what" in clear specification and "how" in efficient implementation, which also allowed us to arrive at a corrected and simplified specification of RBAC compared with the ANSI standard [13].

2. Language

We use a standard imperative language that supports sets, tuples, and maps for the specification and implementation of core RBAC, though an object-oriented language could be used where core RBAC could be just one class, which is why we use the names "field" and "method" below. We use the name "program" for both specifications and implementations. Figure 1 defines the language, where X^*, X^+, and $X^?$ denote that X occurs 0 or more, 1 or more, and 0 or 1 times, respectively.

---

prog ::= (*fieldname* : *type*)*
    (*methodname* (*varname* *): (pre-cond: *expr* ;)$^?$ *stmt*)*
type ::= set(*type*) | tuple(*type* $^+$) | map(*type*, *type*) | ...
stmt ::= for (*varname* in *expr*)$^+$ | *expr* : *stmt* | ...
expr ::= {*expr* : (*varname* in *expr*)$^+$ | *expr* } | ...
fieldname , *methodname* , *varname* : identifiers

Figure 1. Language.

A program defines a set of fields and a set of methods, possibly with pre-conditions. Types may be specified not only for fields but also for variables, method parameters, and return values, although we omit those types from the grammar. We generally omit types when they can be inferred from the program. Note the types for sets, tuples, and maps, the statement for iterating over sets, and the expression for set comprehension; we make substantial use of these, because sets and tuples are well suited for high-level specifications, and maps are essential for efficient implementations.

For the loop for v_1 in $e_1, ..., v_k$ in e_k | e : s, each variable v_i enumerates elements of the set value of expression e_i, and for each combination of values of v_1 through v_k, if the value of Boolean expression e is true, then execute s. We read the entire statement as "for each v_1 in e_1,..., and v_k in e_k such that e, do s". We omit | e when e is true.

For the set comprehension {e_0 : v_1 in $e_1, ..., v_k$ in e_k | e}, each variable v_i enumerates elements of the set value of expression e_i, and for each combination of values of v_1 through v_k, if the value of Boolean expression e is true, then the value of expression e_0 is an element of the resulting set. We read the expression as "the set of e_0 where v_1 is from e_1,..., and v_k is from e_k such that e". We omit | e when e is true.

We use the following kinds of expressions for other operations on sets, tuples, and maps.

{$x_1, ..., x_k$}	a set with elements $x_1, ..., x_k$
[$x_1, ..., x_k$]	a tuple with elements $x_1, ..., x_k$
{[x_1, y_1], ..., [x_k, y_k]}	
	a binary relation, i.e., a set of 2-tuples, i.e., pairs
$S + T$, $S - T$	union and difference, respectively, of sets S and T
S subset T	whether S is a subset of or equal to T
x in S, x notin S	
	whether or not, respectively, x is an element of S
M{x}	image set of key x under map M

We abbreviate and as a comma. We use indentation to indicate scoping. We use // to begin a comment that lasts till the end of the line.

Our language differs from the Z specification language used in the ANSI standard for RBAC in that it is executable—the semantics of a specification corresponds to a straightforward implementation of the specification,

Cost model. The cost model depends on the implementation of sets and maps. We assume that each set is implemented using a hash table, so it takes constant time on average to retrieve an element from a set, test membership of an element in a set, and add and delete an element to and from a set. We assume that each map is implemented as a hash table for the set of keys and a hash table for the image set of each key, so it takes constant time to add and delete a key, locate an image set of a key, retrieve an element from an image set, test membership of an element in an image set, and add and remove an element to and from an image set.

Each operation that involves enumerating elements of sets has a cost factor that is linear in the size of each set enumerated, and is considered expensive; specifically, the construct

$$x_1 \text{ in } S_1, ..., x_k \text{ in } S_k |$$

has a cost factor of $|S_1| * ... * |S_k|$. This gives a simple way of calculating the time complexities.

3. Specification

Core RBAC contains the following sets and relations and the operations on them summarized in Figure 2, explained below.

```
OBJS:     set(Object)    // an operation-object pair
OPS:      set(Operation) // is called a permission.
USERS:    set(User)
ROLES:    set(Role)
PR:       set(tuple(tuple(Operation,Object),Role))
UR:       set(tuple(User,Role))
          // PR subset (OPS * OBJS) * ROLES
          // UR subset USERS * ROLES
SESSIONS: set(Session)
SU:       set(tuple(Session,User))
SR:       set(tuple(Session,Role))
          // SU subset SESSIONS * USERS
          // SR subset SESSIONS * ROLES
```

A system has sets of objects, operations, users, roles, and sessions; their elements are of types Object, Operation, User, Role, and Session, respectively. A operation-object pair, called a permission, denotes an allowed operation on an object. A permission-role pair in PR denotes a permission assigned to a role. A user-role pair in UR denotes a user assigned to a role. A session-user pair in SU denotes a session and the unique user of the session. A session-role pair in SR denotes a session and a role active in the session.

---

administrative commands	add/delete user/role, assign/deassign user, grant/revoke permission
supporting system functions	create/delete session, add/drop active role, check access
review functions	assigned users/roles
advanced review functions	role/user permissions, session roles/perms, role/user ops on obj

---

Figure 2. Functionalities of core RBAC by categories.

Administrative commands. The following operations each adds an element to a set or a relation.

```
AddUser(user):
  pre-cond: user notin USERS;
  USERS = USERS + {user}

AddRole(role):
  pre-cond: role notin ROLES;
```

```
    ROLES = ROLES + {role}

AssignUser(user,role):
  pre-cond: user in USERS, role in ROLES,
            [user,role] notin UR;
  UR = UR + {[user, role]}

GrantPermission(operation, object, role):
  pre-cond: operation in OPS, object in OBJS,
            role in ROLES,
            [[operation,object],role] notin PR;
  PR = PR + {[[operation,object],role]}
```

Deleting an element is symmetric to adding an element, but possibly with two kinds of additional updates. First, if an element is deleted from a set, then from all relations defined using the set, all tuples that contain the deleted element must be deleted. Second, `DeleteUser`, `DeleteRole`, and `DeassignUser` also delete the associated sessions, to satisfy the constraint that a session can have a role active only if the user of the session is assigned that role.

```
DeleteUser(user):
  pre-cond: user in USERS;
  UR = UR - {[user,r]: r in ROLES} //maintain UR
  for s in SESSIONS |               //maintain SESSIONS
    [s,user] in SU:
    DeleteSession(user,s)//DeleteSession defined below
  USERS = USERS - {user}

DeleteRole(role):
  pre-cond: role in ROLES;
  PR = PR - {[[op,o],role]: op in OPS, o in OBJS}
                                    //maintain PR
  UR = UR - {[u,role]: u in USERS} //maintain UR
  for s in SESSIONS, u in USERS |
    [s,u] in SU, [s,role] in SR:    //maintain SESSIONS
    DeleteSession(u,s)
  ROLES = ROLES - {role}

DeassignUser(user, role):
  pre-cond: user in USERS,role in ROLES,
            [user,role] in UR;
  for s in SESSIONS |
    [s,user] in SU,[s,role] in SR://maintain SESSIONS
    DeleteSession(user,s)
  UR = UR - {[user,role]}

RevokePermission(operation, object, role):
  pre-cond: operation in OPS, object in OBJS,
            role in ROLES,
            [[operation,object],role] in PR;
  PR = PR - {[[operation,object],role]}
```

Supporting system functions. `CreateSession` creates a session for a user with an initial set of active roles; it first checks that the user is assigned those roles, and then adds the appropriate elements to SU, SR, and SESSIONS. `DeleteSession` deletes all elements of SU, SR, and SESSIONS that are associated with the session.

```
CreateSession(user, session, ars):
  pre-cond: user in USERS, session notin SESSIONS,
            ars subset AssignedRoles(user);
            // AssignedRoles is defined below
  SU = SU + {[session,user]}
  SR = SR + {[session,r]: r in ars}
  SESSIONS = SESSIONS + {session}

DeleteSession(user, session):
  pre-cond: user in USERS, session in SESSIONS,
            [session,user] in SU;
```

```
  SU = SU - {[session,user]}
  SR = SR - {[session,r]: r in ROLES} // maintain SR
  SESSIONS = SESSIONS - {session}
```

Adding and deleting active roles adds to and deletes from SR, respectively; adding an active role also first checks that the user of the session is assigned that role.

```
AddActiveRole(user, session, role):
  pre-cond: user in USERS, session in SESSIONS,
            role in ROLES, [session,user] in SU,
            [session,role] notin SR,
            role in AssignedRoles(user);
  SR = SR + {[session,role]}

DropActiveRole(user, session, role):
  pre-cond: user in USERS, session in SESSIONS,
            role in ROLES, [session,user] in SU,
            [session,role] in SR;
  SR = SR - {[session,role]}
```

`CheckAccess` checks whether an operation on an object is allowed in a session, i.e., whether there is a role that is active in the session and is assigned the operation-object pair as a permission.

```
CheckAccess(session, operation, object):
  pre-cond: session in SESSIONS,
            operation in OPS, object in OBJS;
  return {r in ROLES | [session,r] in SR,
                       [[operation,object],r] in PR}
         != {}
```

Review functions and advanced review functions. These functions are queries on the sets and relations. Most of them (`AssignedUsers`, `AssignedRoles`, `RolePermissions`, `SessionRoles`, `RoleOperationsOnObject`) are over one relation, i.e., given a value for the left or right component of a relation, find all associated values for the other component in the relation. For example, the first two are review functions defined by:

```
AssignedUsers(role):
  pre-cond: role in ROLES;
  return {u: u in USERS | [u,role] in UR}

AssignedRoles(user):
  pre-cond: user in USERS;
  return {r: r in ROLES | [user,r] in UR}
```

The other ones (`UserPermissions`, `SessionPermissions`, `UserOperationsOnObject`) are over two relations, called a join in database, i.e., given a value for one component of a relation, equate the other component of the relation with one component of a second relation, and find all associated values for the other component of the second relation. Two of them (`RoleOperationsOnObject`, `UserOperationsOnObject`) involve lookup over nested tuples but are otherwise similar to the other functions. All advanced review functions are defined below:

```
RolePermissions(role):
  pre-cond: role in ROLES;
  return {[op,o]: op in OPS, o in OBJS
          | [[op,o],role] in PR}

UserPermissions(user):
  pre-cond: user in USERS;
  return {[op,o]: r in ROLES, op in OPS, o in OBJS
          | [user,r] in UR, [[op,o],r] in PR}

SessionRoles(session):
  pre-cond: session in SESSIONS;
```

```
    return {r: r in ROLES | [session,r] in SR}

SessionPermissions(session):
  pre-cond: session in SESSIONS;
  return {[op,o]: r in ROLES, op in OPS, o in OBJS
          | [session,r] in SR, [[op,o],r] in PR}

RoleOperationsOnObject(role, object):
  pre-cond: role in ROLES, object in OBJS;
  return {op: op in OPS | [[op,object],role] in PR}

UserOperationsOnObject(user, object):
  pre-cond: user in USERS, object in OBJS;
  return {op: r in ROLES, op in OPS
          | [user,r] in UR, [[op,object],r] in PR}
```

4. Transformational method

Straightforward implementations of many operations in core RBAC are inefficient because they involve iterating through sets from scratch. Efficient implementations require that the results of such expensive computations be stored, be retrieved quickly when needed, and be maintained incrementally when the sets that these results depend on are updated. This section describes the transformational method for incrementalizing individual expensive computations. The overall efficiency depends on which expensive computations to incrementalize based on the costs and frequencies of the operations, and will be discussed in the next section.

Determining expensive computations. We consider all operations that are not constant time expensive. These include set comprehensions, loops over sets, a subset test, a set union, and set differences in the core RBAC specification. A set difference, as in `DeleteUser`, `DeleteRole`, and `DeleteSession`,

$$T = T - \{e: x_1 \text{ in } S_1, ..., x_k \text{ in } S_k\}$$

is transformed to an equivalent loop over sets

$$\text{for } x_1 \text{ in } S_1, ..., x_k \text{ in } S_k \mid e \text{ in } T: T = T - \{e\}$$

since only elements of T need to be considered for deletion. The subset test and set union, in `CreateSession`, are transformed similarly, to loops over `ars`. A loop over sets

$$\text{for } x_1 \text{ in } S_1, ..., x_k \text{ in } S_k \mid e: s$$

is transformed to a set comprehension

$$\{[x_1, ..., x_k] : x_1 \text{ in } S_1, ..., x_k \text{ in } S_k \mid e\}$$

plus a simple loop over the result of the comprehension. For example, in the definition of `DeleteUser`,

```
UR = UR - {[user,r]: r in ROLES} //maintain UR
```

is transformed to

```
for r in ROLES | [user,r] in UR:
  UR = UR - {[user,r]}
```

and then to

```
for r in {r: r in ROLES | [user,r] in UR}:
  UR = UR - {[user,r]}
```

The remaining loops are over the results of set comprehensions in `DeleteUser DeleteRole`, `DeassignUser`, and `DeleteSession` (for maintaining UR, PR, SR, and SESSIONS) and over `ars` in `CreateSession` (for subset test and addition to SR); each of them has a cost proportional to size of the desired outcome and thus can not be eliminated. Therefore, set comprehensions are the only remaining expensive computations that could be optimized away.

Figure 3 lists all 16 occurrences of them, where the first column is the containing method, and last column classifies them into 9 different kinds of queries—1x for four kinds of queries over one relation, and 2x for five kinds of queries over two relations.

The time complexities of these queries can be calculated straightforwardly based on the cost model, which then give the time complexities of the methods, as summarized in the second column of Figure 5.

Size notation. We use the following letters for sizes of the respective sets:

```
set:  OBJS  OPS  OPS*OBJS  USERS  ROLES  SESSIONS
size: O     A    P         U      R      S
```

where `A` for operation is adopted from the initial letter of "access" or "action", and `P` equals `A*O` and is the initial letter of "permission". We use `x/y`, where x and y are the letters for the above sets but in lower case, to denote the number of x's per y. For example, `s/u` denotes the number of sessions per user, which can be used either for the worst case or the average case analysis. Specially, `s/ur` denotes the number of sessions per user per active role, and `a/or` denotes the number of operations allowed per object per role.

Identifying parameter updates. The parameters of a query are the free variables in the query. The result of the query depends on the values of these variables. For example, parameters of the `CheckAccess` query are `ROLES`, `SR`, `PR`, `session`, `operation`, and `object`.

An update to a parameter is any operation that sets the value of the parameter. For example, for the `CheckAccess` query, the parameters `ROLES`, `SR`, and `PR` are set by addition and deletion of an element, while `session`, `operation`, and `object` are set by calls to `CheckAccess`.

We identify primitive updates to the parameters, i.e., primitive operations that directly set the values of the parameters, not through method calls. For example, the primitive updates in `DeleteUser` are the assignments to `UR` and `USERS`, not the call to `DeleteSession` that assigns to `SU`, `SR`, and `SESSIONS`. Figure 4 lists all primitive updates to the sets and relations defined in core RBAC, where the first column is the containing method, and the parentheses enclose the number of elements added or deleted if the number is not 1. These sets and relations are all the parameters, besides method arguments, of all the expensive queries.

```
AddUser           add to        USERS
DeleteUser        delete from   USERS, UR (r/u)
AddRole           add to        ROLES
DeleteRole        delete from   ROLES, UR (u/r), PR (p/r)
AssignUser        add to        UR
DeassignUser      delete from   UR
GrantPermission   add to        PR
RevokePermission  delete from   PR
CreateSession     add to        SU, SR (ars), SESSIONS
DeleteSession     delete from   SU, SR (r/s), SESSIONS
AddActiveRole     add to        SR
DropActiveRole    delete from   SR
```

Figure 4. Primitive updates in core RBAC.

Deriving incrementalization rules. For each kind of expensive query, we need to know how to handle each kind of update to a parameter of the query; this is captured by an incrementalization rule, which systematically addresses three main issues in incrementalizing expensive set queries.

containing method	expensive query	kind
DeleteUser	`{r: r in ROLES	[user,r] in UR}`	1a
	`{s: s in SESSIONS	[s,user] in SU}`	1b
DeleteRole	`{[op,o]: op in OPS, o in OBJS	[[op,o],role] in PR}`	1c
	`{u: u in USERS	[u,role] in UR}`	1b
	`{[s,u]: s in SESSIONS, u in USERS	[s,u] in SU, [s,role] in SR}`	2e
DeassignUser	`{s: s in SESSIONS	[s,user] in SU, [s,role] in SR}`	2d
DeleteSession	`{r: r in ROLES	[session,r] in SR}`	1a
CheckAccess	`{r: r in ROLES	[session,r] in SR, [[operation,object],r] in PR}`	2c
AssignedUsers	`{u: u in USERS	[u,role] in UR}`	1b
AssignedRoles	`{r: r in ROLES	[user,r] in UR}`	1a
RolePermissions	`{[op,o]: op in OPS, o in OBJS	[[op,o],role] in PR}`	1c
UserPermissions	`{[op,o]: r in ROLES, op in OPS, o in OBJS	[user,r] in UR, [[op,o],r] in PR}`	2a
SessionRoles	`{r: r in ROLES	[session,r] in SR}`	1a
SessionPermissions	`{[op,o]: r in ROLES,op in OPS,o in OBJS	[session,r] in SR,[[op,o],r] in PR}`	2a
RoleOperationsOnObject	`{op: op in OPS	[[op,object],role] in PR}`	1d
UserOperationsOnObject	`{op: r in ROLES, op in OPS	[user,r] in UR, [[op,object],r] in PR}`	2b

Figure 3. Expensive queries in core RBAC.

First, to handle parameters that can be set to any value, a map is maintained that maps the values of those parameters to the results of the query. For example, for the `CheckAccess` query, we maintain a map, called `MapSP2R`, that maps any given session and permission, i.e., operation-object pair, to the desired set of roles. Then, the `CheckAccess` query can be replaced with a fast retrieval,

```
MapSP2R{[session,operation,object]}
```

which locates the result set in constant time. The method `CheckAccess` can then test the emptiness of the result set in constant time, and thus is overall constant time and optimal.

Second, to handle other parameter updates, we must derive code for incrementally maintaining the result of the query at each kind of update. For example, for the `CheckAccess` query, `MapSP2R` must be incrementally maintained when `ROLES` is updated in `AddRole` and `DeleteRole`; `SR` is updated in `CreateSession`, `DeleteSession`, `AddActiveRole`, and `DeleteActiveRole`; and `PR` is updated in `GrantPermission`, `RevokePermission`, and `DeleteRole`. The derivation starts with generic code for maintaining the result set, obtained from the set comprehension by iterating over both the sets enumerated and the sets tested for membership. For example, the generic code for the `CheckAccess` query is

```
for r in ROLES: for [s,r] in SR: for [[op,o],r] in PR:
    ... //add r to, or delete r from, MapSP2R{[s,op,o]}
```

A variable that becomes bound in an outer loop is used as a filter for the values enumerated in an inter loop. For the example above, the variable `r` becomes bound in the outer-most loop, so for each value of `r`, among tuples enumerated in the two inner loops, only those whose second component equals the value of `r` are considered. The specific code derived for maintaining the result set at each kind of update is described below.

Third, for efficient incremental computation on sets of tuples, auxiliary maps that map values of certain components of the tuples to values of other components of the tuples are needed to quickly retrieve the values of latter components given values of the former components. These maps essentially serve as indices. For the `CheckAccess` query, the generic form above shows the need to find all `s`'s and all `[op,o]`'s that match each `r` in `SR` and `PR`, respectively. So, we maintain two auxiliary maps: `SRMapR2S` maps each role in the range of `SR` to its corresponding set of sessions in `SR`, and

`PRMapR2P` maps each role in the range of `PR` to its corresponding set of permissions in `PR`. That is, `SRMapR2S` and `PRMapR2P` are the inverse maps of `SR` and `PR`, respectively.

Now, to obtain the specific maintenance code at each addition and deletion, we start with the generic code, and do four things: (1) eliminate the loop over the set that is being added or deleted an element, because only the element being added or deleted needs to be considered for this loop in the incremental maintenance, (2) replace each loop whose loop variables are all bound with a test on the loop variables, because bound variables are filters of the values, (3) use auxiliary maps in loops that have both bound and unbound loop variables to iterate over only the values of the unbound variables, and (4) update an auxiliary map when its corresponding set of tuples is updated. We show the resulting maintenance code for the `CheckAccess` query at each kind of update to `ROLES`, `SR`, and `PR`. We first show the cases for element additions, as a set of `at update do maintenance time formula` clauses:

```
at ROLES = ROLES + {r}
do for s in SRMapR2S{r}: for [op,o] in PRMapR2P{r}:
    ... //add r to MapSP2R{[s,op,o]}
time O(s/r*p/r)

at SR = SR + {[s,r]}
do if r in ROLES: for [op,o] in PRMapR2P{r}:
    ... //add r to MapSP2R{[s,op,o]}
   SRMapR2S{r} = SRMapR2S + {s}
time O(p/r)

at PR = PR + {[[op,o],r]}
do if r in ROLES: for s in SRMapR2S{r}:
    ... //add r to MapSP2R{[s,op,o]}
   PRMapR2P{r} = PRMapR2P{r} + {[op,o]}
time O(s/r)
```

where the code for adding `r` to `MapSP2R{[s,op,o]}` is

```
if r notin MapSP2R{[s,op,o]}:
   MapSP2R{[s,op,o]} = MapSP2R{[s,op,o]} + {r}
```

Deletion is symmetric and has the same cost, i.e., is exactly the same except with `if...notin` replaced with `if...in` and with `+` replaced with `-`. Finally, initialization of the maps is simple: the map for the query result is set to empty when any of the sets being

iterated over is set to empty, and an auxiliary map is set to empty when the corresponding set of tuples is set to empty; the cost is always $O(1)$.

Two kinds of simplifications can be made to the maintenance code above for additions in core RBAC, although they do not improve the actual asymptotic running times. First, the maintenance code in the first clause can be eliminated, and its time complexity is more accurately $O(1)$, because the range of `SR` is a subset of `ROLES` in RBAC, and thus `SRMapR2S` maps a new role to the empty image set. Second, in the maintenance code in the second clause, the condition `if r in ROLES` can be removed, again because the range of `SR` is a subset of `ROLES`, and thus the `r` in a pair added to `SR` must be in `ROLES`.

Rules for the four kinds of `1x` queries are much simpler; each of them just needs to store, use, and maintain one auxiliary map. Rules similar to the rule for the `CheckAccess` query can be derived for the other four kinds of `2x` queries. For some queries, different order of sets and relations being iterated may lead to different complexities, because of differences in the order of binding the variables. The number of possible orders is exponential in the number of sets being iterated, but it is typically a small constant, so we can simply consider all of them.

Applying incrementalization rules. Since all queries in core RBAC are independent of each other, rules for incrementalizing them can be applied in any order. There are two additional details when applying individual rules. First, if the maintenance code at a parameter update uses the value of the parameter before the update, then it must be inserted before the parameter update; otherwise, it can be inserted either before or after. Second, even though a query result can be located in constant time, if it is iterated over, then a copy of it needs to be made for the iteration if the query result may be incrementally updated at updates to the query parameter during the iteration; note that this copying does not increase the overall asymptotic running time because the cost of copying is amortized over the subsequent iteration.

5. Efficient implementations

An overall efficient implementation depends on the frequencies and the needed response times of the operations, because there are tradeoffs between the query times and the update times. We describe three variants that make `CheckAccess` efficient, make all queries efficient, and make retrievals over single relations efficient, respectively, some at the expense of increased update times. Other variants that make different sets of queries efficient can be obtained using the same transformational method.

Making `CheckAccess` efficient. Supporting system functions are typically the most frequently executed in all core RBAC functionalities. Among these functions, `CheckAccess` is typically the most frequent, and `CreateSession` and `DeleteSession` are the next most frequent. Straightforward execution of them each takes $O(R)$ time.

Incrementally maintaining the `CheckAccess` query makes `CheckAccess` efficient, as shown in Section 4, but it adds maintenance work at the updates to `SR` in `CreateSession` and `DeleteSession`, so this is a good tradeoff if `CheckAccess` is performed much more frequently than `CreateSession` and `DeleteSession` or if the response time of `CheckAccess` is most critical compared to the other two. Meanwhile, the cost of $O(R)$ in `CreateSession` and `DeleteSession` can be eliminated, by incrementally maintaining the expensive query `AssignedRoles`

that is called in `CreateSession`, and the expensive query in `DeleteSession` that is the same as the query `SessionRoles`.

The third column in Figure 5 summarizes the time complexities of the functionalities in the resulting implementation. Space complexity is the sum of $O(S * P + |PR| + |SR|)$ for storing the maps for making `CheckAccess` constant time, $O(|UR|)$ for making `AssignedRoles` constant time, and $O(|SR|)$ for making `SessionRoles` constant time.

Making all queries efficient. All queries are listed in Figure 3. To make them all efficient, we apply incrementalization rules to all of them. This leads to further increase in the time complexities of the update operations, and so is a good tradeoff if all queries are performed much more frequently than the updates or if the query response time is critical, such as during an intensive policy review period. The fourth column in Figure 5 summarizes the time complexities of the operations in the resulting implementation. Space complexity is the sum of the sizes of all maps maintained for all of the queries.

Making retrievals over single relations efficient. Retrievals over single relations include all queries over one relation, as well as retrievals over one relation in queries over two relations. They can be made efficient by storing, using, and incrementally maintaining auxiliary maps for single relations, not maps that map parameters to results for queries over two relations. These auxiliary maps, determined by retrievals needed in the queries, are all of `URMapR2U`, `URMapU2R`, `SRMapR2S`, `SRMapS2R`, `SUMapS2U`, `SUMapU2S`, `PRMapR2P`, and `PRMapOR2A`.

These auxiliary maps allow the result of queries over single relations to be located in constant time, and queries over two relations to be answered in at most the time proportional to the product of the sizes of the two image sets. For example, the `CheckAccess` query, given the auxiliary map `SRMapS2R`, needs to enumerate only the roles that are active in the given session, rather than all the roles as in the straightforward implementation, and then do a constant-time test against `PR`.

The last column in Figure 5 summarizes the time complexities of the operations in the resulting implementation. While queries over two relations are not as efficient, the update operations are more efficient, compared with when all queries are incrementalized. Indeed, all query and update operations in this implementation are clearly more efficient compared with the straightforward implementation. The space complexities is the sum of the sizes of all auxiliary maps, which is only $O(|UR| + |SR| + |SU| + |PR|)$.

Complexities. Figure 5 lists all core RBAC functionalities and time complexities for their straightforward implementations, and for the three variants discussed above.

6. Experiments

To help confirm the correctness of the transformations and the complexity analysis results presented above, we first developed a straightforward implementation of core RBAC that precisely follows the specification; we then applied our transformational method to it, both manually and automatically, using a number of different combinations of incrementalization rules, and we performed many experiments on the resulting implementations. All experimental results confirmed our expectations.

All implementations were written in Python, which has built-in support for set comprehension; they include the straightforward implementation, several manually incrementalized implementations, and a system we developed to automatically apply incrementalization rules to Python programs. Our current library of incremen-

core RBAC functionalities	straightforward	inc CheckAccess	inc all queries	inc retrievals
AddUser(u)	1			
DeleteUser(u)	R+S+s/u*R	-R+r/u	p/r*(r/u+s/u*r/s)	r/u+s/u*r/s
AddRole(r)	1			
DeleteRole(r)	P+U+S*U+s/r*R	+s/r*p/r+u/r	p/r*(u/r+s/r*r/s)	p/r+u/r+s/r*r/s
AssignUser(u,r)	1		p/r	
DeassignUser(u,r)	S+s/ur*R		p/r*(1+s/ur*r/s)	s/u+s/ur*r/s
GrantPermission(op,o,r)	1	s/r	u/r+s/r	
RevokePermission(op,o,r)	1	s/r	u/r+s/r	
CreateSession(u,s,ars)	ars+R	p/r*ars+r/s	p/r*(ars+r/s)	ars
DeleteSession(u,s)	R	p/r*r/s	p/r*r/s	r/s
AddActiveRole(u,s,r)	R	p/r	p/r	1
DropActiveRole(u,s,r)	1	p/r	p/r	
CheckAccess(s,op,o)	R	1	1	r/s
AssignedUsers(r)	U		1	1
AssignedRoles(u)	R	1	1	1
RolePermissions(r)	P		1	1
UserPermissions(u)	R*P		1	r/u*p/r
SessionRoles(s)	R	1	1	1
SessionPermissions(s)	R*P		1	r/s*p/r
RoleOperationsOnObject(r,o)	A		1	1
UserOperationsOnObject(u,o)	R*A		1	r/u*a/or

Figure 5. Time complexities of core RBAC functionalities. A blank means that it is the same as the straightforward version. A formula prefixed with + or - specifies the difference in the amount from the straightforward version.

talization rules contains nine incrementalization rules, which were developed following the method described in this paper and were expressed using the rule language introduced in [14]. The transformation system, named InvTS (Invariant-driven Transformation System), consists of over 5,000 lines of Python. When applied to the straightforward implementation of core RBAC, it took between 15 to 70 seconds to obtain variants of efficient implementations, depending on the amount of caching used in InvTS, and the number of queries incrementalized in each variant.

We can get a sense of how much effort incrementalization saved us by comparing the size of the straightforward program to the size of the incrementalized ones. We report the number of interesting lines of code, defined as non-empty and non-comment lines. The straightforward program consists of 125 lines of interesting code, including 16 expensive queries that could be incrementally maintained. When all 16 queries are incrementalized, the code more than tripled in size to 486 lines.

We developed a program that lets us perform black-box testing on both an original and an incrementalized implementations, to confirm that they produce the same output. It generates a sequence of random RBAC operations that are applied to both implementations. When an operation produces a result, the results produced by both implementations are compared, and verified to be identical. This lets us automatically test the incrementalized program against the original program it was generated from. All tested implementations produced identical results for a sequence of 50 million operations, giving us confidence in the correctness of the incrementalized implementations.

We developed another program to generate data in a way that is governed by one or more independent variables. We used this program to generate a number of sets of input data, varying in some parameter, for each of which we need to determine the running time of each program. For each particular input and program, we compute the running time by running the program repeatedly on the data until the standard deviation of the set of running times is less than 10 percent of the mean of the set of running times. In all cases, the test programs were run a minimum of 10 times.

Our test programs are single-threaded, and were run under Windows XP SP2 on a dual-processor Athlon XP 2.8Ghz with 2 GB of memory, of which around 750 MB was free when running our programs. All of the experiments, written in Python, were run under ActivePython 2.4 Build 244. This system was also used to run the incrementalizer, which took around 30 seconds to complete the incrementalization of RBAC.

We compare the performance of the straightforward implementation and the second variant of the incrementalized implementations, since it shows the biggest tradeoffs on the most important operations (CheckAccess, CreateSession, DeleteSession) and therefore needs the most discussions for when to use the incrementalized version. We measure the time it takes each implementation to complete 1000 repeats of a simple operation pattern. This pattern consists of the creation of a session with 10 active roles, 1000 random access checks, and the deletion of the session. Figure 5 predicts that the operations and parameters that dominate the asymptotic running time differ between the straightforward and incrementalized implementations. In the straightforward implementation, the asymptotic time of should be linear in the number of roles, as access checks and session creation and deletion are all linear in the number of roles. In the incrementalized implementation, CheckAccess should be constant time, and the asymptotic time should be dominated by the cost of CreateSession and DeleteSession, which is linear in the number of permissions assigned to the roles activated in a session.

Figure 6 compares the performance of the implementations where the number of permissions per session is fixed at 100 and number of roles varies. It shows that the incrementalized implementation is unaffected by the increasing number of roles in the system. The straightforward implementation of CheckAccess is
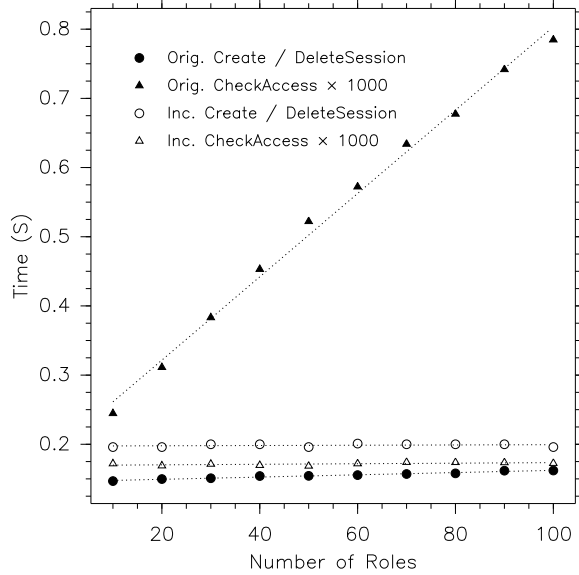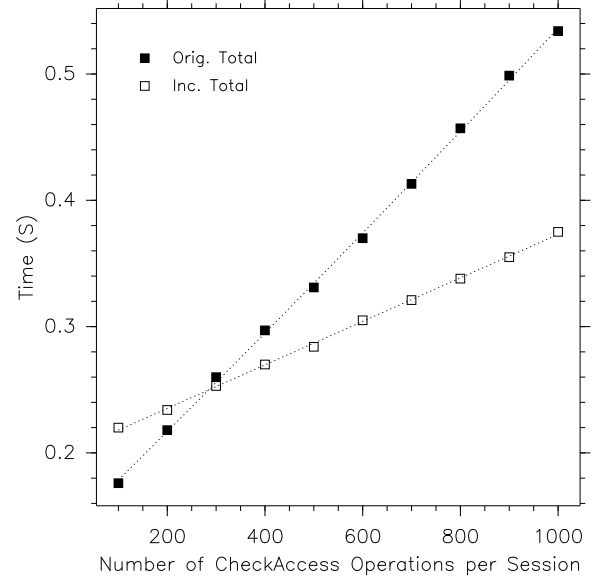
Figure 6. Running time of RBAC operations, 100 permissions per session, 1000 repeats.
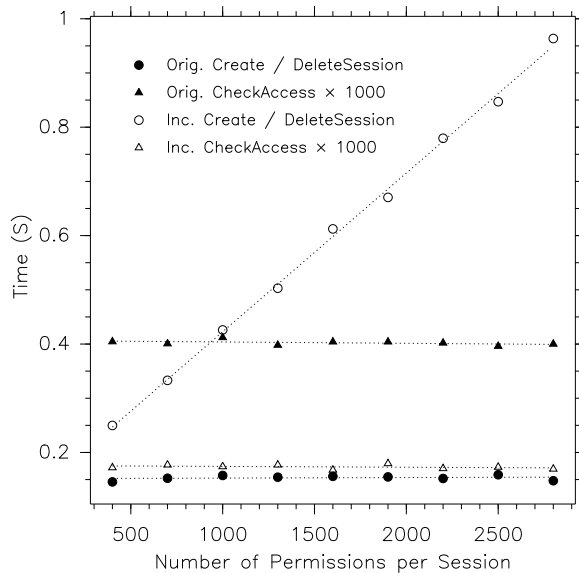
Figure 7. Running time of RBAC operations, 30 roles, 1000 repeats.

linear in the number of roles in the system; so are `CreateSession` and `DeleteSession`, albeit with a much smaller slope, as they occur only once per session, compared to the 1000 times of `CheckAccess`. These conform to the complexity analysis. The total running time of the straightforward version is linear in the number of roles, while the running time of the incrementalized version is constant. This improved asymptotic behavior leads to a practical speedup; with 100 roles, incrementalization improves the total running time from .94 to .37 seconds.

Figure 7 shows the results of a second experiment, where the number of roles in the system is fixed at 30 as the number of permissions per session varies. Again, the results conform to our

Figure 8. Total running time of RBAC, 100 permissions per session, 30 roles, 1000 repeats.

expectations. The asymptotic cost of the incrementalized session creation and deletion increases with the number of permissions per session, while the cost of `CheckAccess` remains constant. The running time of all of the operations in the straightforward version are also asymptotically constant, although the practical cost of the straightforward version of `CheckAccess` is larger than that of the incrementalized version.

These two experiments show that the operations that dominate the running times differ between the two implementations, making it necessary to choose the one that is superior for a given application. When the number of roles and number of permissions per session are fixed, which implementation is faster depends on the number of access checks per session. While both implementations are asymptotically linear in the number of access checks per session, Figure 8 shows that there is a critical number of access checks above which the incrementalized implementation is faster. When this threshold is reached, the decreased cost of the repeated `CheckAccess` operation is enough to pay for the increased cost of `CreateSession` and `DeleteSession`. While the exact value of this critical number will vary with the number of roles and the number of permissions per session, incrementalization will always win for a large enough number of access checks per session.

The incrementalized version is also superior when the cost of access checks is more important than the cost of session creation and deletion. This may be the case in an interactive system in which one is not concerned with the time it takes to create a session, but once that session has been created, demands that operations in it be performed as fast as possible. Here, it makes sense to move costs to non-critical times.

7. Related work and conclusion

The idea of incrementally maintaining the results of set expressions is decades old. In particular, Paige et al studied the subject extensively under the names formal differentiation and finite differencing [16, 19, 18]. Related topics have also been studied by Earley [3], Fong [7, 6], and Yellin and Strom [22], among others, and used for many applications (e.g., [17, 2, 9]). Note that memoization

and dependence graph based techniques can not achieve the same kind of incrementalization, because they do not support specialized auxiliary maps and differential updates. Paige's finite differencing rules are the closest to our incrementalization rules. However, no method was given for systematically deriving the rules. Indeed, developing these rules is the realm of experts and becomes tedious and error-prone when there are many queries that vary slightly from one another.

Liu et al studied a method for incrementalization across object abstraction that is appropriate for implementing core RBAC as a class in the context of larger applications [14]. They also used incrementalization rules without giving the method for deriving the rules. The method described in this paper was taught to graduate students not in programming languages, each deriving rules for some of the expensive queries in core RBAC and applying them to the straightforward implementation, the results of which were then merged straightforwardly though tediously by one of the students, resulting in one of the successful prototypes. Multiple tries by the same group of students before the method was taught failed to produce an efficient implementation with complexity assurance. This helped show that the method is easy to learn and to use by non-experts and is effective. We have also completing the implementation of a prototype system that automatically generates incrementalization rules for queries over sets and objects.

This work is made possible because RBAC has been specified formally and precisely in a set-based specification language, Z [10, 21], in the ANSI standard [1, 5]. The standard defined and incrementally maintained 7 additional maps, in addition to or in place of, the two relations SU and SR defined in Section 3; they appear to be caused by efficiency concerns but complicate the specification unnecessarily. That specification was debugged and simplified in [13], resulting in the specification we used in Section 3. There are many implementations of RBAC (e.g., [23, 15]), but we are not aware of any that provides precise complexity guarantees. Deriving efficient implementations following a systematic transformational method also provides higher assurance for the correctness of the implementations with respect to the specification.

In conclusion, we have developed a systematic method for generating efficient implementations from set-based specifications, and applied it successfully, both manually and automatically, to obtain efficient implementations of core RBAC from a straightforward implementation of the specification based on the ANSI standard. Possible future work includes on-demand computation, i.e., maintaining the query results on demand, as opposed to at all updates to the parameters; other auxiliary data structures beside maps, to possibly improve the query and update times further; efficient methods for considering the order of the sets being iterated; and finally, efficient implementations of hierarchical and constrained RBAC.

References

[1] American National Standards Institute, Inc. Role-Based Access Control. ANSI INCITS 359-2004. Approved Feb. 3, 2004.

[2] B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Science of Computer Programming*, 24(3):189–220, 1995.

[3] J. Earley. High level iterators and a method for automatically designing data structure representation. *J. Comput. Lang.*, 1:321–342, 1976.

[4] D. Ferraiolo and R. Kuhn. Role-based access control. In *Proceedings of the NIST-NSA National Computer Security Conference*, pages 554–563, 1992.

[5] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and Systems Security*, 4(3):224–274, 2001.

[6] A. C. Fong. Inductively computable constructs in very high level languages. In *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages*, pages 21–28, 1979.

[7] A. C. Fong and J. D. Ullman. Inductive variables in very high level languages. In *Conference Record of the 3rd Annual ACM Symposium on Principles of Programming Languages*, pages 104–112, 1976.

[8] A. Gavrila and J. Barkley. Formal specification for RBAC user/role and role relationship management. In *Proceedings of the 3rd ACM Workshop on Role Based Access Control*, pages 81–90, 1998.

[9] D. Goyal and R. Paige. The formal reconstruction and improvement of the linear time fragment of willard's relational calculus subset. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 382–414. Chapman & Hall, London, U.K., 1997.

[10] International Organization for Standardization. Z formal specification notation – Syntax, type system and semantics. ISO/IEC 13568:2002.

[11] T. Jaeger and J. Tidswell. Rebuttal to the NIST RBAC model proposal. In *Proceedings of the 5th ACM Workshop on Role Based Access Control*, pages 66–66, Berlin, Germany, July 2000.

[12] C. E. Landwehr, C. L. Heitmeyer, and J. McLean. A security model for military message systems. *ACM Trans. Comput. Syst.*, 2(3):198–222, 1984.

[13] Y. A. Liu and S. D. Stoller. Role-based access control: A corrected and simplified specification. Technical Report DAR 05-24, Computer Science Department, SUNY Stony Brook, Aug. (Revised Dec.) 2005.

[14] Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, and Y. E. Liu. Incrementalization across object abstraction. In *Proceedings of the 20th ACM Conference Object-Oriented Programming, Systems, Languages, and Applications*, pages 473–486, San Diego, California, Oct. 2005.

[15] National Institute of Standards and Technology. Role-Based Access Control. http://csrc.nist.gov/rbac.

[16] B. Paige and J. T. Schwartz. Expression continuity and the formal differentiation of algorithms. In *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 58–71, 1977.

[17] R. Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In H. Gallaire, J. Minker, and J.-M. Nicolas, editors, *Advances in Database Theory, Vol. 2*, pages 171–209. Plenum Press, New York, Mar. 1984.

[18] R. Paige. Symbolic finite differencing—Part I. In N. D. Jones, editor, *Proceedings of the 3rd European Symposium on Programming*, volume 432 of *LNCS*, pages 36–56. Springer-Verlag, Berlin, 1990.

[19] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, July 1982.

[20] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control: Towards a unified standard. In *Proceedings of the 5th ACM Workshop on Role-Based Access Control*, pages 47–63, Berlin, Germany, July 2000.

[21] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.

[22] D. M. Yellin and R. E. Strom. INC: A language for incremental computations. *ACM Trans. Program. Lang. Syst.*, 13(2):211–236, Apr. 1991.

[23] C. Zhao, Y. Chen, D. Xu, N. Heilili, and Z. Lin. Integrative security management for web-based enterprise applications. In *Proceedings of the 6th International Conference on Web-Age Information Management (WAIM 2005)*, volume 3739 of *LNCS*, pages 618–625–138. Springer-Verlag, Berlin, 2005.