Davide Cecchini

ARBAC Analyser Report

Goal

The challenge requires the realization of an analyser of little ARBAC policies, in particular it must solve the role reachability problem:

Definition

Given an initial user-to-role assignment UR, a policy $\mathcal{P}$ and a role r_g, the role reachability problem amounts to checking whether there exists a user-to-role assignment UR' such that $UR \rightarrow_{\mathcal{P}}^{*} UR'$ and $r_g \in UR'(u)$ for some user u.

Role reachability problem definition from the slides of the course security2 of Ca' Foscari University

For the original assignment text, check arbaclab.pdf.

Pseudocode

The analyzer visit recursively all possible configuration (a configuration is a UR, represent user-to-role assignment) applying all possible rules (CA and CR) to visited configuration and check if one of the new configurations contains an assignment to the goal_role
It keeps track of all visited configurations in order to avoid useless additional computation on already visited configurations.

```
visited_config = initial_UR
current_config = visited_configuration
while( (no one of current_config contains goal_role) and (current_config not
empty) )
    new_config = {}
    for each config in current_config
        new_config_tmp = generate all possible new configuration applying all
possible rules to the config
        for each config2 in new_config_tmp
            if (config2 not in visited_config)
                new_config = new_config + config2
                visited_config = visited_config + config2


    current_config = new_config

if (any of the current_config contains goal_role)
    return true
else
    return false
```

Backward and forward slicing

I applied the backward and the forward slicing techniques to see the improvement of their usage on the program performance (in terms of time spent).

Backward

```
S = {goal_role}
while (S change)
    for each rule in ca
        if r_t in S
            S = S union {r_a} union r_p union r_n

for each rule in ca
    if r_t not in S
        remove rule from ca
for each rule in cr
    if r_t not in S
        remove rule from cr
roles = S
```

Forward

```
S = roles in ur
while (S change)
    for each rule in ca
        if r_t in S
            S = S union {r_a} union r_p union r_n

for each rule in ca
    for each r_p_role in r_p
        if (r_p_role not in S)
            remove rule from ca
for each rule in cr
    if (r_a not in S) or (r_t not in S)
        remove rule from cr
for each rule in ca
    for each r_n_role in r_n
        if (r_n_role not in S)
            remove r_n_role from r_n

roles = S
```

Code

As programming language, I used python.
In the code I used ua instead of ur to refer to the user-to-role .

Function that creates a dict user:roles

```
# used to obtain a dict (<user>:<roles>) from ua
def build_dict_from_pairs(pairs):
    ret_dict = {}
    for pair in pairs:
        if pair[0] in ret_dict:
            ret_dict[pair[0]].append(pair[1])
        else:
            ret_dict[pair[0]] = [pair[1]]
    return ret_dict
```

Function that, giving a configuration and a ca rule, returns the list of users on which the rule can be apply

```
#config is ua, return list of users that can be assigned the r_t of ca
def can_apply_ca(config, ca):
    #check if ra exist
    ra_finded = False
    for ua_pair in config:
        if (ua_pair[1] == ca[0]):
            ra_finded = True
    if (not ra_finded):
        return []
    #build a user:roles dict
    ua_dict = build_dict_from_pairs(config)
    #find users which satisfy r_p and r_n
    users_satisfy = []
    for user in ua_dict:
        flag = True
        #check if user satisfy r_p
        for r_p in ca[1]:
            if (r_p not in ua_dict[user]):
                flag = False

        #check if user satisfy r_n
        for r_n in ca[2]:
            if (r_n in ua_dict[user]):
                flag = False

        #check if user already has r_t
        if (ca[-1] in ua_dict[user]):
            flag = False

        if flag:
            users_satisfy.append(user)
    return users_satisfy
```

Function that, giving a configuration and a cr rule, returns the list of users on which the rule can be apply

```
#config is ua, return list of users that can be revoked the r_t of cr
def can_apply_cr(config, cr):
    #check if ra exist
    ra_finded = False
    for ua_pair in config:
        if (ua_pair[1] == cr[0]):
```

```
            ra_finded = True
    if (not ra_finded):
        return []
    else:
        # for each pair <user,role>, check if role is r_t, if true, return user
        ret_users = []
        for pair in config:
            if pair[1] == cr[1]: ret_users.append(pair[0])
        return ret_users
```

Function that, giving a list of configurations and a goal role, checks if one of them contains the goal role

I test a list of configurations rather than a single configuration because I'll use this function to test all the configurations generated during an iteration (I'll call this function into the while condition).

```
def reached_goal_multiple(configurations, goal_role):
    for config in configurations:
        for user_role in config:
            if (user_role[1] == goal_role): return True
    return False
```

Parse policy from input

```
import sys

roles = []
users = []
ua = []
cr = []
ca = []
goal = ""

for line in sys.stdin:
    if (line[0:5] == "Roles"):
        roles = line.split(" ")[1:-1]
    elif (line[0:5] == "Users"):
        users = line.split(" ")[1:-1]
    elif(line[0:2] == "UA"):
        for ua_pair in line.split(" ")[1:-1]:
            ua.append([ua_pair.split(",")[0][1:],ua_pair.split(",")[1][:-1]])
    elif(line[0:2] == "CR"):
        for cr_pair in line.split(" ")[1:-1]:
            cr.append([cr_pair.split(",")[0][1:],cr_pair.split(",")[1][:-1]])
    elif(line[0:2] == "CA"):
        for ca_tuple in line.split(" ")[1:-1]:
            splitted_tuple = ca_tuple.split(",")
            r_a = splitted_tuple[0][1:]
            r_t = splitted_tuple[2][:-1]
            r_p = []
            r_n = []
            if (splitted_tuple[1] != "TRUE"):
                for r in splitted_tuple[1].split("&"):
                    if (r[0] == "-"):
                        r_n.append(r[1:])
                    else:
```

```
                        r_p.append(r)
            ca.append([r_a,r_p,r_n,r_t])
    elif(line[0:4] == "Goal"):
        goal = line.split(" ")[1]
```

Apply backward slicing

```
#apply backward slicing
backward_states = [goal] #initialize S_0

flag = True
while(flag):
    new_backward_states = backward_states
    for ca_tuple in ca:
        #check if the roles r_t of the ca is in backward_states (S_i-1)
        if (ca_tuple[-1] in backward_states):
            new_backward_states = list(set(new_backward_states) | {ca_tuple[0]}
| set(ca_tuple[1]) | set(ca_tuple[2]))
    if (new_backward_states != backward_states):
        backward_states = new_backward_states
    else:
        flag = False

#remove from ca rules that assign a role in R\S*
ca_new = []
for ca_tuple in ca:
    if ca_tuple[-1] in backward_states:
        ca_new.append(ca_tuple)
ca = ca_new

#remove from cr rules cr_tuplethat revoke a role in R\S*
cr_new = []
for cr_tuple in cr:
    if cr_tuple[-1] in backward_states:
        cr_new.append(cr_tuple)
cr = cr_new

roles = backward_states
```

Apply forward slicing

```
#apply forward slicing

#initialize S_0
forward_states = []
for ua_pair in ua:
    forward_states.append(ua_pair[1])

forward_states = list(set(forward_states))

flag = True
while(flag):
    new_forward_states = forward_states
    for ca_tuple in ca:
        #check if the roles in r_p and r_a of the ca are in forward_states (S_i-
1)
```

```
        inclusion_satisfied = True
        for role in ( [ca_tuple[0]] + ca_tuple[1] ):
            if not(role in forward_states):
                inclusion_satisfied = False
        if (inclusion_satisfied):
            new_forward_states = list(set(new_forward_states) | {ca_tuple[-1]})
    if (set(new_forward_states) != set(forward_states)):
        forward_states = new_forward_states
    else:
        flag = False

#remove from ca rules that include in r_p a role in R\S*
ca_new = []
for ca_tuple in ca:
    flag = True
    for rp_role in ca_tuple[1]:
        if not (rp_role in forward_states):
            flag = False
    if flag:
        ca_new.append(ca_tuple)
ca = ca_new

#remove from cr rules that mention a role in R\S*
cr_new = []
for cr_tuple in cr:
    if cr_tuple[0] in forward_states and cr_tuple[1] in forward_states:
        cr_new.append(cr_tuple)
cr = cr_new

#remove the role R\S* from the negative preconditions of all rule
for ca_tuple in ca:
    rn_new = []
    for rn_role in ca_tuple[2]:
        if rn_role in forward_states:
            rn_new.append(rn_role)
    ca_tuple[2] = rn_new

roles = forward_states
```

Check all possible configurations applying all rules to each config

I use a timeout to limit the execution time in case of negative instances, it is a naif solution since the program will not be stopped exactly 10 seconds after the timeout is setup.

```
import time

visited_configurations = []
visited_configurations.append(set(tuple(item) for item in ua))
current_configurations = []
current_configurations.append(ua)

timeout = time.time()+10
start_time = time.time()
while (not reached_goal_multiple(current_configurations,goal) and
current_configurations and time.time() < timeout):
    new_current_configurations = []
    for config in current_configurations:
```

```
        #for each configuration apply all possible ca
        for ca_to_apply in ca:
            users_to_apply = can_apply_ca(config, ca_to_apply)
            #for each ca apply it to all possible user
            for user in users_to_apply:
                #for each user apply ca adding the new role
                new_config = config.copy()
                new_config.append([user,ca_to_apply[-1]])
                #if the configuration is new, add it to the configuration to
visit
                if set(tuple(item) for item in new_config) not in
visited_configurations:
                    new_current_configurations.append(new_config)
                    visited_configurations.append(set(tuple(item) for item in
new_config))

        for cr_to_apply in cr:
            users_to_apply = can_apply_cr(config, cr_to_apply)
            #for each cr apply it to all possible user
            for user in users_to_apply:
                #for each user apply cr removing the role
                new_config = config.copy()
                new_config.remove([user,cr_to_apply[-1]])
                #if the configuration is new, add it to the configuration to
visit
                if set(tuple(item) for item in new_config) not in
visited_configurations:
                    new_current_configurations.append(new_config)
                    visited_configurations.append(set(tuple(item) for item in
new_config))
    current_configurations = new_current_configurations

print(time.time()-start_time)
print(reached_goal_multiple(current_configurations,goal))
```

Results

I ran the program (without slicing) on the first policy (returning true and terminating without visiting all possible configs), I noticed that the execution time was too great: 781 sec.
Using forward slicing I obtained slight (and not satisfying) improvements: 736 sec.
With backward slicing the situation totally changes, with an execution time of 0.3 sec.
The results obtained using other two policies (policy3 and policy4) are similar:

Policy	Without slicing	Forward slicing	Backward slicing	Backward and Forward
Policy1	781.3073	736.3435	0.2361	0.2646
Policy3	0.6556	0.4759	0.0028	0.0023
Policy4	871.9254	722	7.0803	5.0134

execution time (in seconds) of the program on different policies using or not slicing

So i decided to use backward slicing to complete the task and I test other policies obtaining the flag: 10110110.
I tried to use backward and forward slicing together and as you can notice, there is not a great improvement and, considering the time spent to perform the additional slicing, for little policies

we can be satisfied using only the backward slicing.