LANGUAGE-BASED TECHNOLOGY FOR SECURITY

Master of Science in Cybersecurity

Homework Assignment 2

Extending the Interpreter with History-Dependent Access Control and Recursion

Professor: Gian Luigi Ferrari

Students: Giuseppe Crea, Francesco Venturini, Laura Norato, Marco Parti

Academic Year 2020/2021

Introduction

This homework will extend the interpreter of the simple functional language into an interpreter of a functional language which includes a dynamic mechanism to enforce local security policies along the lines of History-Dependent Access Control.

Local security policies are characterized by a scope and are defined over a set of security relevant actions. The interpreter checks the policy within its scope and continues execution if the policy is satisfied; otherwise, execution fails.

The decision was made to continue the implementation of the interpreter from our first homework assignment. The OCaml file contains lines of code enclosed by two kinds of comments:

- **(* Homework 1 *)** ***code*** **(* /Homework 1 *)** to identify the code relative to the Stack Inspection implementation from the first assignment.

- **(* Homework 2 *)** ***code*** **(* /Homework 2 *)** to identify the code relative to the History-Dependent Access Control implementation from the new assignment.

This choice was made with the intention of building a single large project starting from individual assignments.

Deterministic Finite Automata

The implementation of the *DFA* is almost the same as the one provided by the slides of the assignment.

```
33  (* Homework 2 *)
34
35  type state = int;;
36
37  type symbol = char;;
38
39  type transition = state * symbol * state;;
40
41  type dfa =    {
42    states : state list;
43    sigma : symbol list;
44    start : state;
45    transitions : transition list;
46    accepting : state list;
47  };;
48
49  (* /Homework 2 *)
```

A *DFA* is a 5-tuple, consisting of:

- A finite set of states, implemented as int
- A finite set of input symbols (alphabet), implemented with OCaml's built-in char type
- A transition function δ : state * symbol -> state, implemented as a list of tuples
- A starting state
- A set of accepting states as a list of states

Since these automata define the policies that a programmer can implement in our language, we must first define a finite set of automata, containing all possible policies that a programmer can use.

To manage *DFAs* we decided on a solution based on the class construct of OCaml:

```
(* Class for managing DFAs *)
class dfa_container =

  object (self)

    val mutable dfa_list = ([]: dfa list)
    val mutable securityHistory = (""): string)  (* Global variable holding the value of the execution history *)

    method push dfa =
      dfa_list <- dfa :: dfa_list

    method pop = (* pop method *)
      let result = List.hd dfa_list in
      dfa_list<- List.tl dfa_list;

    method extendHistory c =  (* static method that will add char c to string securityHistory *)
      securityHistory <- securityHistory^c;

    method viewHistory =
      securityHistory

    method checkSecurityHistory =
      checkSecurityHistoryHelper dfa_list securityHistory

    method clearHistory =
      securityHistory <- ""

  end;;
```

The class *dfa_container* is composed by two mutable fields:

- *dfa_list*: the field will hold the list of currently active automata within our scope
- *securityHistory:* a string of symbols of the alphabet of our automata, representing every single security operation that has been executed by the code so far

And six methods:

- *push dfa*: adds the dfa automaton to the top of the dfa list
- *pop*: drops the automaton currently on top of the dfa list
- *extendHistory*: adds a symbol to the execution history
- *viewHistory*: returns the complete execution history (unused)
- *checkSecurityHistory*: calls a recursive helper method:
    - *checkSecurityHistoryHelper*: calls *checkAccepts* with the current security history string on every automaton in the dfa list; returns true if and only if the computation is accepted on every single one of those automata

```
let rec checkSecurityHistoryHelper dfa_list history : bool=
  match dfa_list with
  | [] -> true
  | hd::tl -> if checkAccepts history hd then checkSecurityHistoryHelper tl history else false
```

- *clearHistory*: resets the local history, used when loading a new program

The main advantages of this solution are:

- the use of private mutable fields for the dfa list and the security history
- gathering together every method and data structure needed for the implementation of this homework, improving readability

The instantiation of the class is the object called *dfa_overlord* in the program.

Updating of ieval for Local Security Policies

The added changes were applied on the operations that are relevant for security.

```
(* currently identical dfas are allowed *)
| Phi(dfa, e) ->
    dfa_overlord#push dfa;
    let interval = ieval e env g pStack in
    dfa_overlord#pop;
    interval

| Read(_) ->
    dfa_overlord#extendHistory "r";
    if dfa_overlord#checkSecurityHistory then
      if stackWalking pStack P_Read then (Int 0, g) else failwith("No Read permission on stack")
    else failwith("Illegal History.")

| Write(_) ->
    dfa_overlord#extendHistory "w";
    if dfa_overlord#checkSecurityHistory then
      if stackWalking pStack P_Write then (Int 0, g) else failwith("No Read permission on stack")
    else failwith("Illegal History.")

| Send(x, _) ->
    dfa_overlord#extendHistory "s";
    if dfa_overlord#checkSecurityHistory then
      if stackWalking pStack P_Send then let (_, g') = ieval x env g pStack in (Int 1, g') else failwith("No Send permission on stack")
    else failwith("Illegal History.")
```

- *Phi(dfa, e)*: a new operation used to represent security framing of a block. The operation takes a dfa that is added to the top of the *dfa_list* of the object *dfa_overlord*, through the push method, executing the framing-in action. After that, *ieval* is called to evaluate the expression *e* in env, and the result is saved in the variable interval. Then the dfa is removed from the top of the dfa list with the pop method, which executes the framing-out, and the intervalue is returned.
  Since *Phi* is a new construct, it was also added to *expr*, *iexpr*, and *comp*.

- *Read, Write, Send*: these operations have been modified to include the requested functionalities of History-Dependent Access Control. Through the method *extendHistory* each of these operations adds a symbol to the security history, and through the method *checkSecurityHistory* the new security history is evaluated.

Recursive Functions Implementation

The recursive function implementation proposed in earlier slides has been slightly modified to account for our global environment and permission stack.

```
LetRec(f, i, fBody, letBody, permList) ->
  let (rval, g') = ieval (FunRec(f,i,fBody, permList)) env g pStack in
  let benv = (f,rval)::env
  in ieval letBody benv g' pStack
```

Unlike the normal *Let* construct, this case can only be called when a recursive function is being declared. Thus, it will need a permission list, like all functions, and it will also need to evaluate the new global state *g'* obtained from the evaluation of the recursive closure of the function body, which

is then bound to the function name f on the environment. Finally, just like the proposed implementation, the body of the function is evaluated in this new environment.

```
| RecClosure (fName, x, fBody, fDeclPermList, fDeclEnv) ->
    let (xVal, g') = ieval eArg env g pStack in
    let rEnv = (fName, fClosure) :: fDeclEnv in
    let fBodyEnv = (x, xVal) :: rEnv in
    let fDeclPermStack = fDeclPermList::pStack
    in ieval fBody fBodyEnv g' fDeclPermStack
```

The *RecClosure* has similarly been modified to accommodate for the expanding of the permission stack, and the changing global variable.

```
  | FunRec(f, x, fBody, permList) -> (RecClosure (f, x, fBody, permList, env), g);
```

FunRec returns a *RecClosure* element and a global state element.

Testcases

As part of the language the programmer is offered a sample automaton called *noWaR*, a security policy in which no Write operations can be done after a Read operation, a sort of *Reverse Chinese Wall Policy*. This is equivalent to having an API for requesting a specific policy.

Extended from last homework, by adding the Phi call on top of the short code we are calling. This will execute a whole code as a security block with the policy that is implemented by the automaton that is passed as argument to Phi.

Just like Homework 1, the *goodTest* expression is designed to always complete an eval call, while the *badTest* is designed to always fail. Unlike Homework 1, this time the permissions are identical, and the only difference is in the order in which Read and Write are called. This is used to show the correctness of the History-Dependent Access Control mechanism.

The recursion testcase is taken from the slides and adapted to our language.