Language-based Technology for Security
Homework 1

Lorenzo Gazzella - 546890

April 29, 2022

1 Definition of the programming language

The chosen programming language over which implement the operator that allows the secure execution of mobile code is a functional language. Below we can find the core of the language:

```
type exp =
  | Eint of int
  | Ebool of bool
  | Var of ide
  | BinOp of string * exp * exp
  | UnOp of string * exp
  | If of exp * exp * exp
  | Let of ide * exp * exp
  | Fun of ide * exp
  | FunCall of exp * arg
  | Letrec of ide * exp * exp
  | Execute of exp * permission list
  | Send of string * exp
  | Receive of string * int
  | Write of string * exp
  | Read of string * int
```

It is a classical functional language with the addition of some operators:

1. **Send/Receive:** They are dummy operators that represent the send/receive of a value on/from a given socket.
2. **Write/Read:** They are similar to the two operators above but they use files instead of sockets.
3. **Execution:** This is the operator that runs the mobile code. It is explained in more detail in the next section.

2 Execution operator

To introduce properly the execution operator we have to clarify some aspects:

- In the proposed implementation, the execution operator can be recursive i.e. we can have that execution can run another execution. If we think of a real scenario, there could be the case that we want to execute a mobile code, and this one calls in turn another mobile code.
- The operator takes an expression (i.e. the mobile code) and a permission list. The latter is a list of all the rights that the current execution provides to the child execution. We can have 5 kinds of permission:

  1. **PMemory:** Right of using memory, both on writing and on reading
  2. **PWrite:** Right of writing something on a file
  3. **PRead:** Right of reading something from a file
  4. **PSend:** Right of using the socket to send something
  5. **PReceive:** Right of using the socket to receive something

  Notice that the last 4 permissions respect a sort of decreasing monotony. It means that if an execution wants to run a child execution, the latter one cannot have some permission that the current does not have. This is because in some sense we are using the same resources as the parent. To clarify we can see an example:

  Example Suppose that we run a mobile code `mc1` with permission list [`PRead`], so that he can access and read files. Then suppose that `mc1` runs a mobile code `mc2` with permission list [`PRead`; `PWrite`]. If we would permit `mc2` to write on a file, we will violate the inital permission allowed to `mc1`.

  In the other hand, the permission `PMemory` does not work like the other ones. In fact the idea of the memory management is that:

  1. If the parent execution allows the child to use its memory, then the latter can see every declaration made by the parent. In terms of code, we call the `eval` function to evaluate the mobile code with the same environment
  2. Otherwise, we have to create a separate environment in which the mobile code can create its own declarations but cannot access any other ones.

  Hence, if we have the same scenario as before where we have `mc1` and `mc2`, we have to allow a configuration where `mc1` has zero permissions and `mc2` has [`PMemory`]. In this way, we have conceptually two memory: the one of which only the partent execution can access, and a fresh memory that will be shared between `mc1` and `mc2`.

3 Trusted execution

To implement the design choice discussed before, the `eval` function now takes three parameters:

1. The expression that it has to evaluate
2. The environment that is has to use
3. The set of permission that the current expression has

The evaluation of `Send`, `Receive`, `Write`, `Read` are quite trivial. They only have to check if there is its permission on the permissions list. The evaluation of `Execution` has to do two tasks:

1. Check if the permission list of the mobile code, looking only at the ones that regard sockets and files, is included in the permission list of the current execution
2. Check if it has to start the evaluation of the mobile code in a new environment

4 Example

Concluding, we look to an example of code:

```
let e4 = Let("x", Eint(1), Execute(Let("y", Eint(5),
    Execute(Write("./file.txt", BinOp("-", Var("x"), Var("y"))),
    [PWrite; PMemory])), [PWrite; PMemory]));;}
```

that in an ocaml-like code it is:

```
let x = 1 in Execute(let y = 5 in Execute(Write("./file.txt", x-y)))
```

Suppose we call the outer Execution `E1` with its permission list `PL_E1` and the inner one `E2` with `PL_E2`. Let's now see how the output changes when we change the permission list of both.

- `PL_E1 = PL_E2 = [PWrite; PMemory]`: The result would be the writing of x-y on the file since `E2` has the write permission and it can access to both x and y.
- `PL_E1 = [PWrite; PMemory] and PL_E2 = [PWrite]`: The result would be an error that evidence the fact that `E2` cannot evaluate the expression x-y since it does not share the memory with `E1` and so it cannot access either x or y.
- `PL_E1 = [PWrite] and PL_E2 = [PWrite; PMemory]`: The result would be an error that is arisen from the access to x when it wants to evaluate

x-y. Note that in this case y is readable since `E1` and `E2` share the memory. The problem that we cannot access x raise up from the fact that the outer execution is started with a fresh environment.