Laboratory for Innovative Software

Giulio Paparelli

27/03/2023

The provided interpreter is for a simple functional programming language that supports basic operations such as arithmetic, logical, and comparison operations, as well as let bindings, conditionals, functions, recursive functions, and higher-order functions.

The interpreter is based on an abstract syntax tree (AST) that is defined using the expr type. The interpreter evaluates an expression in the AST by traversing the tree and computing the result of the expression. The result of the evaluation is a triple that consists of the value of the expression, the environment in which the expression was evaluated, and the taintness status of the value.

There is also a built-in mechanism for propagating taintness through the program. The taintness status is introduced by the stub function **GetInput**, and then the taintness of a value is propagated through the program by tracking the taintness status of each expression in the AST.

More in detail, there is support for the following expressions:

- **EInt**: to represent an integer;
- **EBool**: to represent a boolean;
- **EChar**: to represent a character;
- **Var**: to represent an identifier, a variable;
- **LetIn**: to represent Ocaml's *let-in* command;
- **If**: to represent the conditional statement;
- **Prim**: to represent binary operations;
- **Fun**: to represent the definition of a function;
- **FunR, Letrec**: to represent the definition of a recursive function;
- **Call**: to represent the application of a function;
- **GetInput**: stub function to represent the input that the user may provide;

Consequently, the **value** type can represent characters, integers, booleans, and (closures of) functions and recursive functions.

The **eval** function that implements the interpreter takes two parameters:

- **expr**: the expression to be evaluated;
- **env**: the environment;

And returns a triplet $\langle$*computed value*, *env*, *taintness of the computed value*$\rangle$

Every input from the function **GetInput** is considered tainted. The taintness is then propagated by the application of binary operations. The execution is never aborted, even when it is based on tainted values. It may be safer to abort such computations (a taint condition in an **if** statement can lead to unwanted behavior, e.g.: the attacker can impose the execution of a given branch), but this way it is possible to fully check the taintness propagation, and besides it would be straightforward to change the interpreter to abort dangerous computations.

The nature of this functional programming language makes easier the taint analysis as there is no support for locations or jumps.

Finally, the interpreter was tested with simple examples to check that the evaluation of

- **higher order functions**
- **tainted let assignment**
- **untainted let assignment**
- **function application on tainted arguments**

would be as expected.