



UNIVERSITY OF PISA
MASTER'S DEGREE IN CYBERSECURITY

REPORT FOR LANGUAGE BASED
TECHNOLOGY FOR SECURITY PROJECT

COURSE LBTS

Author(s)
Conti Simone
Galli Gabriele
Pini Samuele

Academic year 2023/2024

Contents

1 Homework Specifications	1
2 Abstract Syntax Tree	3
2.1 Design of the AST	3
2.1.1 Values and taintness	4
3 Environment and Taint analysis	6
3.1 Design of the Environment Module	6
3.1.1 Type Definitions	6
3.1.2 Environment Operations	7
3.2 Taint Management	8
3.3 Conclusion	8
3.4 Taint Analysis	8
3.5 Taint Analysis in <code>taint.ml</code>	8
3.5.1 <code>is_tainted</code> Function	8
3.5.2 <code>mark_tainted</code> Function	9
3.6 Conclusion	9
4 Utilities	10
4.1 Introduction	10
4.2 Design of the utilities module	10
4.2.1 Function implementation	10
4.3 Conclusion	13
5 Interpreter	14
5.1 Design of the Interpreter Module	14
5.1.1 Core <code>eval</code> Function	14
5.2 Conclusion	21
6 Tests	22
6.1 <code>BinOp</code> 's test	22
6.2 If tests	23

6.3	plugin tests	24
6.4	Execute tests	24
6.5	Trust block tests	25
6.6	Secrets Tests	26

CHAPTER 1

Homework Specifications

In this project, the task is to design a simple functional language with primitive abstractions to protect the execution of programs from untrusted code.

The assignment involves implementing a trusted interpreter for a functional language in OCaml. This language includes linguistic primitives that manage the secure execution of untrusted code. It provides a programming model with abstractions for declaring trusted code and data, as well as mechanisms for invoking trusted code and executing external plugins.

For example, consider the following trusted block of code:

```
let trust mycode =
trust{
  let secret string password = "abcd";
  let checkPassword (guess: string): bool =
    password = guess;
  handle checkPassword;
}
```

In this example, the trust abstraction allows the declaration of both trusted functions and trusted data within a secure block. The *checkPassword* function, annotated with *handle*, serves as the interface between the trusted block and the external environment, ensuring that secret data, such as the password, is not leaked.

Additionally, the language supports including and executing untrusted plugins provided by external suppliers. For instance, the following code demonstrates the inclusion of a plugin that filters elements in a list:

```
let myFilter = include {  
  filter : ('a -> bool) -> 'a list -> 'a list = <fun>;  
}
```

```
let even n = n mod 2 = 0 in  
execute(myFilter, even [1; 2; 3; 4])
```

The execution of this plugin produces the list [2; 4]. However, the attacker controls the plugin code, and it is crucial to prevent the release of sensitive values, such as passwords.

To ensure the integrity of interactions between trusted and untrusted code, the language's execution engine uses dynamic taint analysis. This technique controls the flow of tainted data within the program, preventing data leakages.

This report covers two main contributions:

- Discussing the language design to support trust programming and plugin inclusion.
- Describing the implementation of its execution environment (interpreter, dynamic taint tracking, and runtime support) and evaluating its usage through simple case studies.

CHAPTER 2

Abstract Syntax Tree


In this chapter we present the design and implementation of the Abstract Syntax Tree (AST) in OCaml for our language.

2.1 Design of the AST

The AST is defined in the `ast.ml` file as shown below in *Figure 2.1*. The `expr` type represents the allowed expressions in our language:

Each variant in the `expr` type corresponds to a different kind of expression:

- `CstInt`, `CstBool`, and `CstStr` represent integer, boolean, and string constants.
- `CstList` represents a list of expressions.
- `BinOp` represents binary operations, defined by an operator and two operand expressions.
- `Var` represents a variable identified by an identifier.
- `Let` represents a let-binding like in OCaml.
- `If` represents conditional expressions.
- `Fun` and `Call` represent function definitions and function calls, respectively.
- `Trust` represent trusted code blocks. `Handle` is used for secure execution within a trusted context.
- `Include` and `Exec` represent the inclusion and execution of plugins.
- `Secret` represents secret data that must be protected from untrusted code.



```


1  type expr =
2    | CstInt of int
3    | CstBool of bool
4    | CstStr of string
5    | CstList of expr list
6    | BinOp of ide * expr * expr
7    | Var of ide
8    | Let of ide * expr * expr
9    | If of expr * expr * expr
10   | Fun of ide * expr
11   | Call of expr * expr
12   | Trust of expr * expr
13   | Handle of expr
14   | Include of ide * (value env -> value)
15   | Exec of expr * expr list
16   | Secret of string

```

Figure 2.1: *expr* type in AST

2.1.1 Values and taintness

The value type, shown in *Figure 2.2*, represents the possible values that expressions can evaluate to.



```

1  and value =
2    | Int of int
3    | Bool of bool
4    | String of string
5    | ListVal of value list
6    | Closure of ide * expr * value env
7    | Taintedval of taintness
8    | SecretVal of string
9
10 and taintness = taint * value

```

Figure 2.2: *value* type in AST

Each variant in the value type corresponds to a different kind of runtime value:

- Int, Bool and String represent constant integer, boolean and string values like the corresponding ones in expr (with a different name).

-
- `ListVal` represents lists of values.
 - `Closure` represents function closures, which include the function parameter, body, and the environment in which the function was defined.
 - `Taintedval` represents values that may have been tainted by untrusted code, encapsulating both the taint status and the actual value.
 - `SecretVal` represents secret values that should not be leaked.

The `taintness` type pairs a `taint` status with a value, where:

- `Tainted` indicates that the value has been influenced by untrusted code.
- `Untainted` indicates that the value is safe and has not been influenced by untrusted code.

CHAPTER 3

Environment and Taint analysis

This chapter discusses the design and implementation of the environment module in OCaml, defined in the `env.ml` file. The environment module manages the mappings between variable identifiers and their corresponding values. This mechanism supports the secure execution of both trusted and untrusted code.

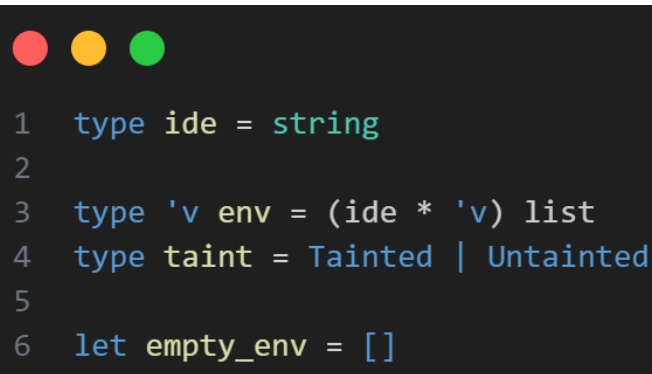
3.1 Design of the Environment Module

The `env` module provides the fundamental operations for handling variable bindings, which include looking up variable values, extending the environment with new bindings, and initializing an empty environment. It introduces the concept of taint, which is essential for tracking and managing the security status of data.

3.1.1 Type Definitions

The `env.ml` defines the types used within the environment as shown in *Figure 3.1* :

- `ide` is an alias for `string`, representing variable identifiers.
- `'v env` is a type alias for a list of pairs, where each pair consists of an identifier and its associated value.
- `taint` represents the taint status of a value, indicating whether it is `Tainted` or `Untainted`.
- `Empty_env` initializes an empty environment



```
1 type ide = string
2
3 type 'v env = (ide * 'v) list
4 type taint = Tainted | Untainted
5
6 let empty_env = []
```

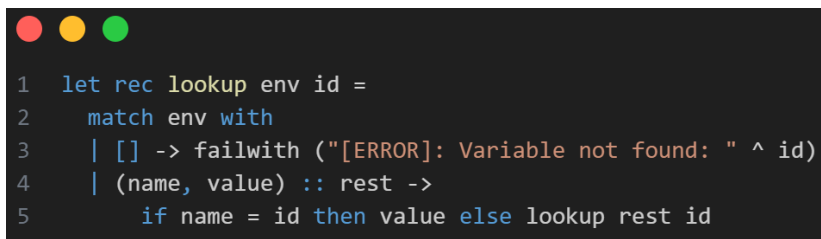
Figure 3.1: *type definition in env*

3.1.2 Environment Operations

The environment module provides several key functions for manipulating environments:

Lookup

The `lookup` function retrieves the value associated with a given identifier from the environment as shown by *Figure 3.2*:



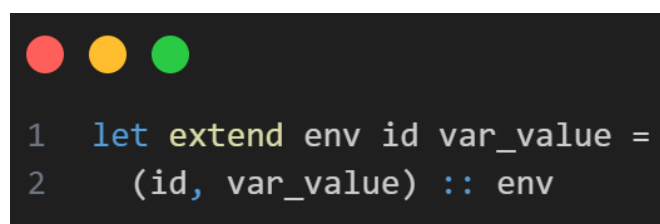
```
1 let rec lookup env id =
2   match env with
3   | [] -> failwith ("[ERROR]: Variable not found: " ^ id)
4   | (name, value) :: rest ->
5     if name = id then value else lookup rest id
```

Figure 3.2: *lookup function*

This function recursively searches through the environment list. If the identifier is not found, it raises an error which tell us which identifier was not found in the environment.

Extend

The `extend` function adds a new binding to the environment as shown in *Figure 3.3*:



```
1 let extend env id var_value =
2   (id, var_value) :: env
```

Figure 3.3: *extend function*

This function creates a new environment by adding the new binding to the existing environment list.

3.2 Taint Management

The environment module also adds the concept of taint management. The `taint` type, defined as `Tainted` or `Untainted`, is used to track whether data has been influenced by untrusted code. This is necessary to prevent security breaches and to ensure the integrity of the program's execution.

3.3 Conclusion

The `env.ml` file defines the essential operations for managing variable bindings within the interpreter for the functional language. The inclusion of taint management further enhances the security by tracking the flow of potentially unsafe data. This module ensures that the language can handle both trusted and untrusted code, maintaining the integrity and security of the execution environment.

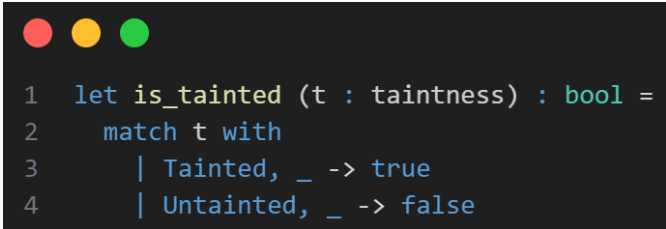
3.4 Taint Analysis

This section explores the implementation of taint analysis and secure execution mechanisms within the `taint.ml` module of an OCaml interpreter. Specifically, we will see how the functions are designed to handle taintness of values and ensure secure operations within a trusted computing environment.

3.5 Taint Analysis in `taint.ml`

3.5.1 `is_tainted` Function

The `is_tainted` function is used to determine whether a given value is tainted or untainted based on the `taintness` type.



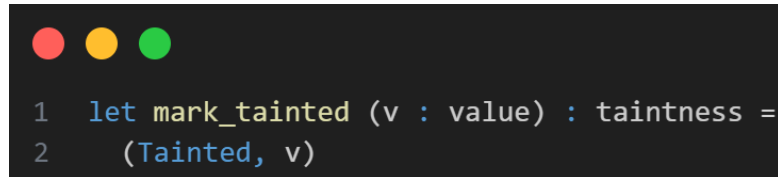
```
1 let is_tainted (t : taintness) : bool =
2   match t with
3     | Tainted, _ -> true
4     | Untainted, _ -> false
```

Figure 3.4: *is_taint function*

This function pattern matches on the `taintness` tuple, returning `true` if the value is marked as `Tainted` and `false` if it is `Untainted`. It provides a clear mechanism to query the security status of data elements.

3.5.2 `mark_tainted` Function

This function facilitates the explicit marking of a value as tainted. This operation is crucial for ensuring that sensitive data is handled with the appropriate level of caution within the interpreter.

A screenshot of OCaml code in a dark-themed editor. At the top left, there are three colored circles: red, yellow, and green. Below them, the code is as follows:

```
1 let mark_tainted (v : value) : taintness =  
2   (Tainted, v)
```

Figure 3.5: *mark_tainted* function

By wrapping the value `v` in a tuple `(Tainted, v)`, this function effectively signifies that the associated data is now considered tainted.

3.6 Conclusion

The `taint.ml` module plays a critical role in enforcing secure programming and taint analysis within the OCaml interpreter. By implementing functions like `is_tainted`, `mark_tainted` it enables developers to control the flow of sensitive data and maintain integrity during program execution.

In subsequent chapters, we will explore how we decided to implement the taint analysis and for what we believed was useful.

CHAPTER 4

Utilities

4.1 Introduction

This chapter discusses the design and implementation of the utilities module in OCaml, defined in the `utilities.ml` file. This module provides a number of utilities for printing, string handling, and checking secret values. The main objective of these functions is to facilitate the management and manipulation of data within the user agent, while ensuring security and proper display of information.

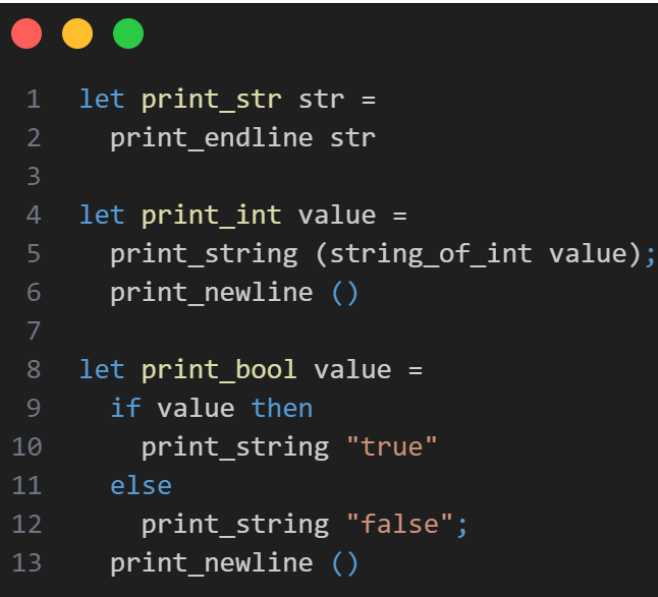
4.2 Design of the utilities module

While most of the function present in this module are used for debugging purposes, there are some function which are fundamental for the protection of the secret variables.

4.2.1 Function implementation

print functions

- The `print_str` function is useful to print the value of the `str` variable. Useful for debugging purposes.
- The `print_int` function is useful to print the value of the `int` variable. It converts the value of the `int` variable to a `str` and then prints it to the screen. Useful for debugging purposes.
- The `print_bool` function is useful to print the value of the `bool` variable. It prints the value of the variable as a string. Useful for debugging purposes.

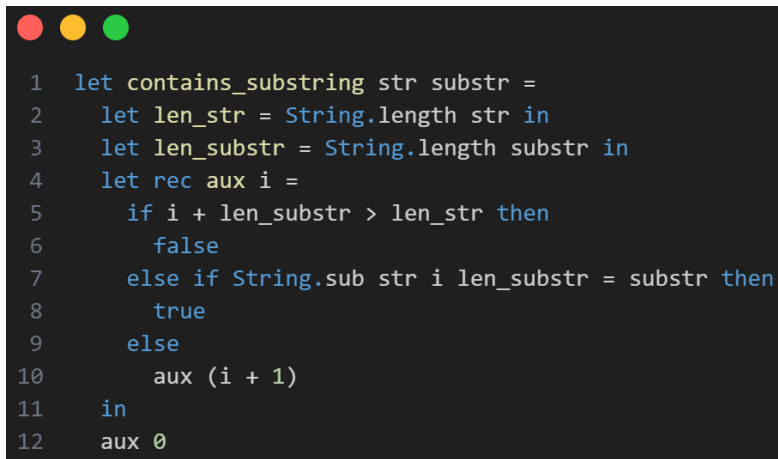


```
1 let print_str str =
2   print_endline str
3
4 let print_int value =
5   print_string (string_of_int value);
6   print_newline ()
7
8 let print_bool value =
9   if value then
10    print_string "true"
11  else
12    print_string "false";
13  print_newline ()
```

Figure 4.1: *print functions in utilities.ml*

contains_substring function

This function checks whether a string `substr` is contained in another string `str`. It is implemented using a recursive aux function that runs through the main string checking if the substring is present. This function is crucial for operations that require checking for the presence of specific patterns in text strings. It is currently used in the main to check if the expected exception from a test is the one required. It could be used for other purposes if needed.



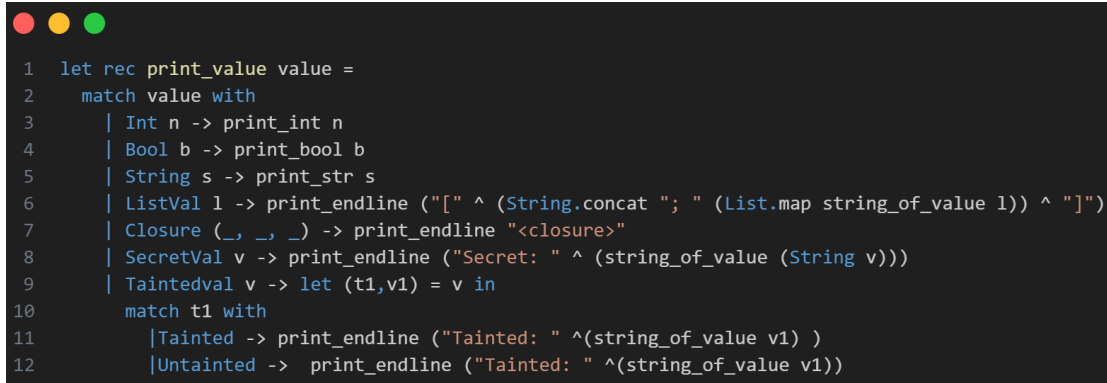
```
1 let contains_substring str substr =
2   let len_str = String.length str in
3   let len_substr = String.length substr in
4   let rec aux i =
5     if i + len_substr > len_str then
6       false
7     else if String.sub str i len_substr = substr then
8       true
9     else
10      aux (i + 1)
11   in
12   aux 0
```

Figure 4.2: *contains_substring function in utilities.ml*

print_value function

This recursive function prints a value based on its type. It supports various data types such as integers, Booleans, strings, lists, closures (`Closure`), secret values (`SecretVal`),

and tainted values (`Taintedval`). This function is essential for the structured display of interpreter results.

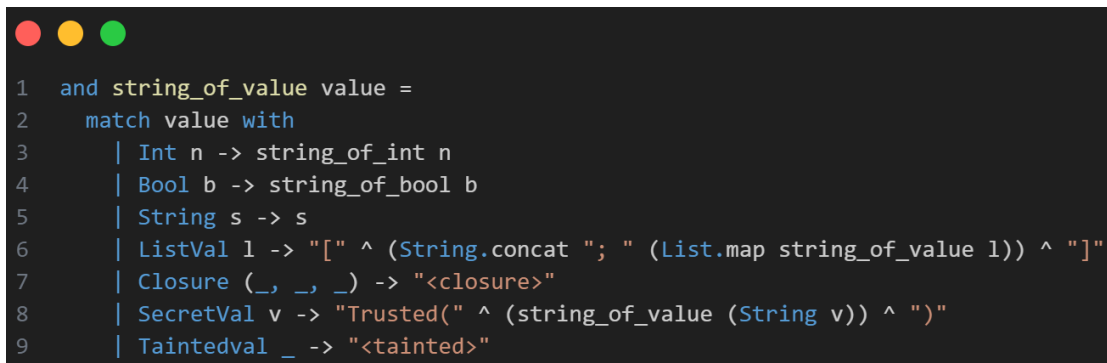


```
1 let rec print_value value =
2   match value with
3   | Int n -> print_int n
4   | Bool b -> print_bool b
5   | String s -> print_str s
6   | ListVal l -> print_endline ("[" ^ (String.concat "; " (List.map string_of_value l)) ^ "]")
7   | Closure (_, _, _) -> print_endline "<closure>"
8   | SecretVal v -> print_endline ("Secret: " ^ (string_of_value (String v)))
9   | Taintedval v -> let (t1,v1) = v in
10    match t1 with
11    | Tainted -> print_endline ("Tainted: " ^ (string_of_value v1) )
12    | Untainted -> print_endline ("Tainted: " ^ (string_of_value v1))
```

Figure 4.3: *print_value* function in *utilities.ml*

string_of_value function

This function converts a value to a representative string. It is similar to `print_value` but returns a string instead of printing it directly. This function is useful when a string representation of values is needed for further processing. It is used with the `print_value` function to format the string based on the value itself. Both this function and `print_value` are mainly used for debugging purposes.

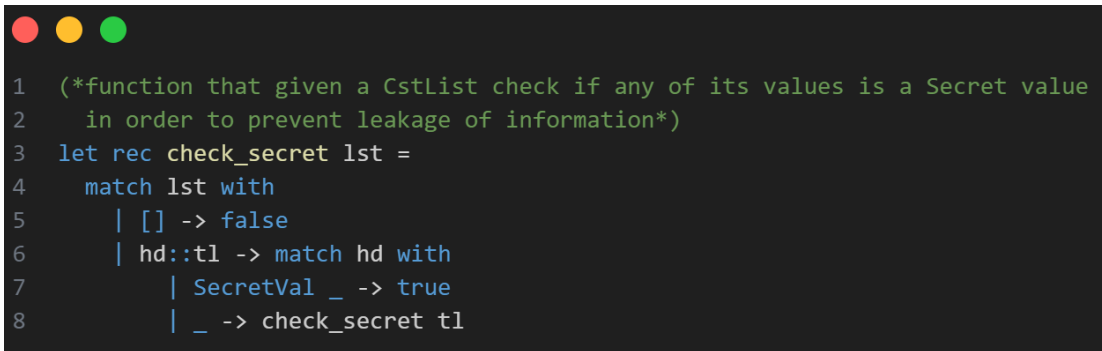


```
1 and string_of_value value =
2   match value with
3   | Int n -> string_of_int n
4   | Bool b -> string_of_bool b
5   | String s -> s
6   | ListVal l -> "[" ^ (String.concat "; " (List.map string_of_value l)) ^ "]"
7   | Closure (_, _, _) -> "<closure>"
8   | SecretVal v -> "Trusted(" ^ (string_of_value (String v)) ^ ")"
9   | Taintedval _ -> "<tainted>"
```

Figure 4.4: *string_of_value* function in *utilities.ml*

check_secret function

This function checks whether a list of values (`CstList`) contains a secret value (`SecretVal`). It is implemented recursively, examining each element in the list. This function is crucial to prevent the loss of sensitive information. The actual usage of this function will be explained in more details in the chapter 5 where we will explain the implementation of the interpreter with the reasoning behind every choice.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays OCaml code for a recursive function named `check_secret`. The code is as follows:

```
1  (*function that given a CstList check if any of its values is a Secret value
2     in order to prevent leakage of information*)
3  let rec check_secret lst =
4      match lst with
5      | [] -> false
6      | hd::tl -> match hd with
7          | SecretVal _ -> true
8          | _ -> check_secret tl
```

Figure 4.5: *check_secret* function in *utilities.ml*

4.3 Conclusion

The *utilities.ml* file represents a set of essential tools for the OCaml interpreter. The implemented functions not only facilitate data management and visualization, but also ensure the security of sensitive information. The modular approach and clear separation of responsibilities make this code easy to maintain and extend, contributing to the overall robustness of the interpreter.

CHAPTER 5

Interpreter

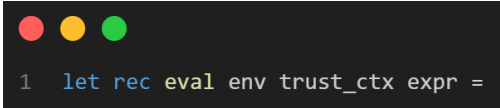
This chapter provides a detailed discussion on the design and implementation of the interpreter module in OCaml, as defined in the `interpreter.ml` file. The interpreter is responsible for evaluating expressions in the functional language, managing the interplay between trusted and untrusted code, and ensuring secure execution through dynamic taint analysis.

5.1 Design of the Interpreter Module

The interpreter module is structured around the core `eval` function, which recursively evaluates expressions within a given environment and trust context. The design leverages pattern matching extensively to handle different types of expressions.

5.1.1 Core `eval` Function

The `eval` function is defined as follows:



```
1 let rec eval env trust_ctx expr =
```

Figure 5.1: *eval function in interpreter.ml*

This function takes three parameters:

- `env`: the current environment, mapping variable identifiers to their values.
- `trust_ctx`: a boolean indicating whether the current context is trusted or untrusted.

- `expr`: the expression to be evaluated.

The `eval` function uses pattern matching to determine the type of expression and apply the appropriate evaluation logic.

Constants

These patterns shown in *Figure 5.2* handle constant values (integers, booleans, strings, and lists). The list pattern recursively evaluates each element in the list.

```

1  match expr with
2  | CstInt n -> Int n
3  | CstBool b -> Bool b
4  | CstStr s -> String s
5  | CstList l -> ListVal (List.map (eval env trust_ctx) l)

```

Figure 5.2: matching of the constant in the `eval` function

Binary Operations

This pattern shown in *Figure 5.3* handles binary operations, which include arithmetic operations and comparisons. The inner `match` expression checks the types of operands and performs the operation if they are integers, strings, bools or raises an error otherwise. The result is checked for taint propagation.

```

1  | BinOp (op, e1, e2) -> (
2    let v1 = eval env trust_ctx e1 in
3    let v2 = eval env trust_ctx e2 in
4    begin
5      (match (v1, v2) with
6      | (Int n1, Int n2) -> (
7        let res =
8          match op with
9          | "+" -> Int (n1 + n2)
10         | "-" -> if n1 < n2 then failwith "[ERROR]: Negative results are forbidden" else Int (n1 - n2)
11         | "/" -> if n2 = 0 then failwith "[ERROR]: Division by zero not allowed" else Int (n1 / n2)
12         | "*" -> Int (n1 * n2)
13         | "=" -> Bool (n1 = n2)
14         | "%" -> Int (n1 mod n2)
15         | _ -> failwith ("[ERROR]: Operation not recognized: " ^ op)
16       in
17         match v1, v2 with
18         | Taintedval _, _ | _, Taintedval _ -> Taintedval (Tainted, res)
19         | _ -> res)
20      | (String s1, String s2) -> (
21        let res =
22          match op with
23          | "=" -> Bool (s1 = s2)
24          | "+" -> String (s1 ^ s2)
25          | _ -> failwith ("[ERROR]: Unknown primitive operation for strings: " ^ op)
26        in
27          match v1, v2 with
28          | Taintedval _, _ | _, Taintedval _ -> Taintedval (Tainted, res)
29          | _ -> res)
30      | (Bool b1, Bool b2) -> (
31        let res =
32          match op with
33          | "||" -> Bool (b1 || b2)
34          | "&&" -> Bool (b1 && b2)
35          | _ -> failwith "[ERROR]: Unknown operator or wrong types for operation"
36        in
37          match v1, v2 with
38          | Taintedval _, _ | _, Taintedval _ -> Taintedval (Tainted, res)
39          | _ -> res)
40      | _ -> failwith ("[ERROR]: Type error in primitive operation: " ^ op))
41    end
42  )

```

Figure 5.3: matching of binary operations in the `eval` function

As shown in the *Figure 5.3* we decided to also check for division by zero errors. Additionally, we decided to not allow subtraction leading to negative numbers because, for instance, if we have code involving some kind of balance and the programmer does not check that the user has enough currency to perform the operation, it may result in a negative balance, which is highly undesirable. While this may be seen as an extreme constraint (we are essentially removing any operation that may change the sign of a value), it could be viewed as a way to reduce the need for programmers to check for logic errors in applications that use budget types with purchases, thereby preventing potential logic errors.

Variables

This pattern shown in *Figure 5.4* handles variable lookup. If the context is trusted, accessing a tainted value raises an error to prevent potential security breaches.

```
1 | Var v -> (  
2   let var_val = lookup env v in  
3   if trust_ctx then  
4     match var_val with  
5     | Taintedval _ -> failwith "[ERROR]: Attempted access to tainted value within trust block"  
6     | _ -> var_val  
7   else  
8     var_val  
9 )
```

Figure 5.4: matching of var in the eval function

Let Bindings

This pattern shown in *Figure 5.5* handles `let` bindings by evaluating the value expression, extending the environment with the new binding, and then evaluating the body expression. Taint is propagated if necessary.

```
1 | Let (id, val_expr, body_expr) -> (  
2   let let_value = eval env trust_ctx val_expr in  
3   let new_env = extend env id let_value in  
4   (* Propagate taint to the newly created environment *)  
5   let new_env =  
6     match let_value with  
7     | Taintedval _ -> extend new_env "let_result" (Taintedval (Tainted, let_value))  
8     | _ -> new_env  
9   in  
10  eval new_env trust_ctx body_expr  
11 )
```

Figure 5.5: matching of let in the eval function

Conditionals

This pattern shown in *Figure 5.6* handles `if` expressions by evaluating the condition and branching based on its boolean value.

```

1 | If (cond, e1, e2) -> (
2   let v_cond = eval env trust_ctx cond in
3   begin
4     match v_cond with
5     | Bool true -> eval env trust_ctx e1
6     | Bool false -> eval env trust_ctx e2
7     | _ -> failwith "[ERROR]: Condition of If must be of a boolean value"
8   end
9 )

```

Figure 5.6: *matching of if in the eval function*

Functions and Function Calls

The `Fun` pattern shown in *Figure 5.7* creates a closure with the function body and environment. The `Call` pattern evaluates the function and argument expressions, then applies the function by extending its environment with the argument and evaluating the body.

```

1 | Fun (param, fbody) -> Closure (param, fbody, env)
2
3 | Call (eFun, eArg) -> (
4   let fCall = eval env trust_ctx eFun in
5   let arg_val = eval env trust_ctx eArg in
6   begin
7     match fCall with
8     | Closure (param, fBody, fEnv) ->
9       (* Propagate taint to function arguments *)
10      let new_env =
11        match arg_val with
12        | Taintedval _ -> extend fEnv param (Taintedval (Tainted, arg_val))
13        | _ -> extend fEnv param arg_val
14      in
15      eval new_env trust_ctx fBody
16    | _ -> failwith "[ERROR]: Expected a closure in Call"
17  end
18 )

```

Figure 5.7: *matching of Fun and Call in the eval function*

Secret

The *Secret* pattern shown in *Figure 5.8* returns the `SecretVal` of the string `s`.

```

1 | Secret s -> SecretVal s

```

Figure 5.8: *matching of Secret in the eval function*

Trust Blocks

This pattern shown in *Figure 5.9* handles trust blocks by evaluating the trusted code in a trusted context and ensuring that no tainted values are introduced. The result is checked for taint before returning. In this pattern matching we make sure that we are not allowing the declaration of a trust block inside another trust block. This can be done because when we are evaluating a trust block code we change the trust context to trusted.

We also check that the variables passed to the trust block are not Tainted, because we only want to have trusted variables inside the trust block.

```
1 | Trust (tCode, tBody) -> (  
2   if trust_ctx then  
3     failwith "[ERROR]: Trust block cannot be declared inside a trust block"  
4   else  
5     (* Check if the value provided in the trust block is tainted *)  
6     let tVal = eval env true tCode in  
7     let new_env = match tVal with  
8       | SecretVal s -> extend env "trusted" (SecretVal s)  
9       | TaintedVal _ -> failwith "[ERROR]: Tainted value provided in a trust block"  
10      | _ -> extend env "trusted" tVal  
11     in  
12     let res = eval new_env true tBody in  
13     (* Check if the value returned from the trust block is tainted *)  
14     match res with  
15       | TaintedVal _ -> TaintedVal (Tainted, res)  
16       | _ -> res  
17 )
```

Figure 5.9: matching of trust block in the eval function

Handle Blocks

This pattern shown in *Figure 5.10* manages handle, ensuring they are only used within trusted contexts and do not contain exec calls, which are highly not trusted, coming from an external source. The result is a closure that is evaluated in its environment.

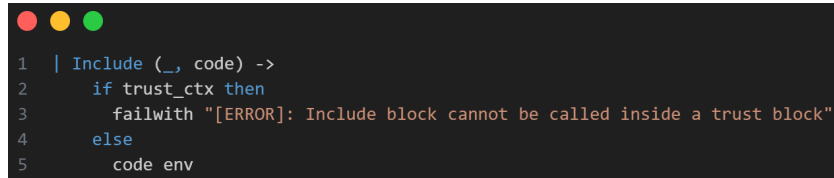
```
1 | Handle expr -> (  
2   if not trust_ctx then  
3     failwith "[ERROR]: Handle block cannot be called outside a trust block"  
4   else  
5     match expr with  
6     | Exec (_,_) -> failwith "[ERROR]: Handle block cannot contain an Exec call"  
7     | _ -> let hVal = eval env trust_ctx expr in  
8         match hVal with  
9         | Closure (_, body, fEnv) ->  
10            (* Recursively evaluate the closure body within the closure's environment *)  
11            eval fEnv trust_ctx body  
12         | _ -> failwith "[ERROR]: Expected a closure in handle"  
13 )
```

Figure 5.10: matching of handles in the eval function

Include

This pattern shown in *Figure 5.11* manages include types. From the requirements of the project we were told that we cannot inspect the code of the plugin. For this

reason we decided to just execute the code of the plugin itself, without using the eval function. We are also checking that we are not trying to include a plugin inside a trust block, because we only want trusted code inside a trust block, and not code which can be controlled by an attacker.



```
1 | Include (_, code) ->
2   if trust_ctx then
3     failwith "[ERROR]: Include block cannot be called inside a trust block"
4   else
5     code env
```

Figure 5.11: *matching of include in the eval function*

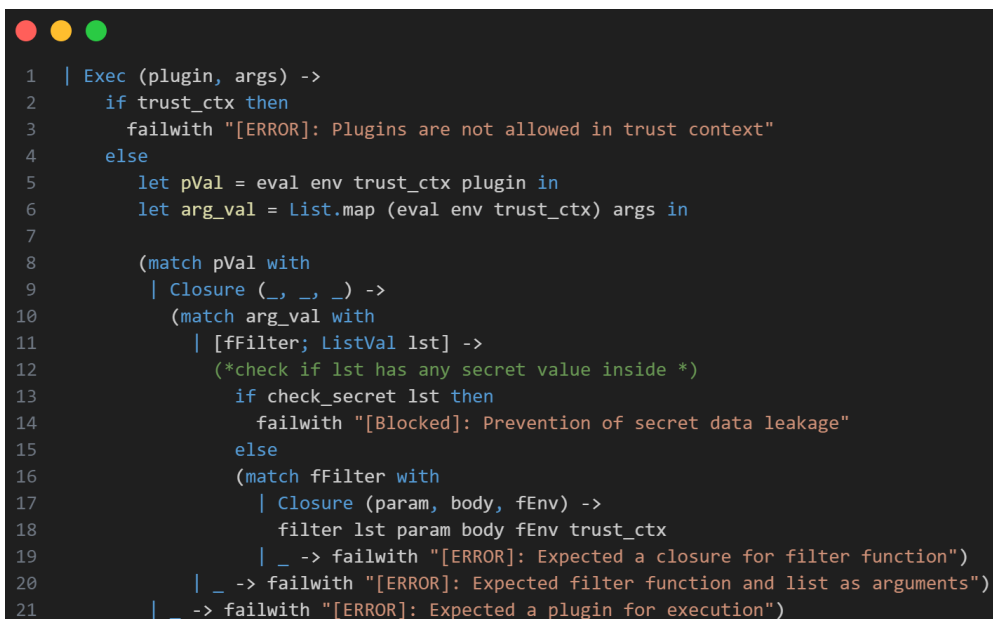
Exec call

This pattern shown in *Figure 5.12* manages `exec` calls. We first check that we are not executing in a trusted context, this is another check just to make sure that we are not inside a trust block. We then evaluate the plugin itself and its arguments. We then check that we have a closure as a plugin otherwise we throw an error.

During this phase we had to take a decision on what kind of plugin we decided allow. After careful consideration we decided to only allow filter functions, this is because if we allowed any types of plugins we could have opened ways for arbitrary code execution, which is highly undesirable (remember that plugins are under the control of an attacker, we have no way to prevent them from doing harm, we see the plugins as a blackbox).

After matching the filter and its list of arguments we then check that the arguments does not match a secret variable declared in a trusted block. If it does we block its execution to avoid the leakage of the secret from the execution of the filter function.

If it does not match any secret variable we then just execute the filter.



```
1 | Exec (plugin, args) ->
2   if trust_ctx then
3     failwith "[ERROR]: Plugins are not allowed in trust context"
4   else
5     let pVal = eval env trust_ctx plugin in
6     let arg_val = List.map (eval env trust_ctx) args in
7
8     (match pVal with
9       | Closure (_, _, _) ->
10         (match arg_val with
11           | [fFilter; ListVal lst] ->
12             (*check if lst has any secret value inside *)
13             if check_secret lst then
14               failwith "[Blocked]: Prevention of secret data leakage"
15             else
16               (match fFilter with
17                 | Closure (param, body, fEnv) ->
18                   filter lst param body fEnv trust_ctx
19                 | _ -> failwith "[ERROR]: Expected a closure for filter function")
20           | _ -> failwith "[ERROR]: Expected filter function and list as arguments")
21         | _ -> failwith "[ERROR]: Expected a plugin for execution")
```

Figure 5.12: *matching of exec in the eval function*

The `filter` function shown in *Figure 5.13* applies a predicate to each element in a list and create a new list containing only those elements for which the predicate is true. The function operates recursively, evaluating each element of the list and building the filtered list as it goes.

```

1  and filter lst param body fEnv trust_ctx =
2    match lst with
3    | [] -> ListVal []
4    | h :: t ->
5      let cond = eval (extend fEnv param h) trust_ctx body in
6      (match cond with
7      | Bool true ->
8        (match filter t param body fEnv trust_ctx with
9        | ListVal t_filtered -> ListVal (h :: t_filtered)
10       | _ -> raise (Failure "Expected a ListVal result from filter function"))
11      | Bool false -> filter t param body fEnv trust_ctx
12      | _ -> raise (Failure ("Expected a boolean result in filter function, but got " ^ (string_of_value cond))))

```

Figure 5.13: *filter* function used by *exec*

The parameters of the function are the following:

- `lst`: The list of values to be filtered.
- `param`: The formal parameter that will be used within the body of the predicate.
- `body`: The body of the predicate, expressed as an expression that will be evaluated to determine whether an item should be included in the filtered list.
- `fEnv`: The function environment that provides the context for the evaluation execution.
- `trust_ctx`: The trust context used during expression evaluation, which may affect data access or other security properties.

If the list is empty, the function returns an empty list (`ListVal []`). This is the basic case of recursion. If the list is not empty, the function takes the first element (`h`) and the rest of the list (`t`). The predicate (`body`) is evaluated by extending the function environment (`fEnv`) with the current parameter (`param`) associated with the first element of the list (`h`). If the predicate evaluation results in a boolean value `true`, the current element (`h`) is included in the filtered list. The filter function is called recursively on the rest of the list (`t`). The result of the recursive call is checked to be a list (`ListVal`). If it is a list, the current element (`h`) is added at the beginning of the resulting filtered list. Otherwise, an exception is raised. If the predicate evaluation results in `false`, the current element (`h`) is excluded and the filter function is called recursively on the rest of the list (`t`). If the predicate result is not a boolean value, an exception is raised indicating that a boolean result was expected. This implementation of filter ensures that only elements that satisfy the specified predicate are included in the resulting list, while maintaining a high level of safety and correctness.

env_string and value_string functions

The functions `env_string` and `value_string` shown in *Figure 5.14* are functions used for debugging purposes. They format the messages to be printed based on the type of value that they receives.



```
1 and env_string env =
2   let binds = List.map (fun (name, value) -> name ^ " = " ^ (value_string value)) env in
3   "{" ^ (String.concat "; " binds) ^ "}"
4
5 and value_string value =
6   match value with
7   | Int n -> string_of_int n
8   | Bool b -> string_of_bool b
9   | String s -> s
10  | ListVal l -> "[" ^ (String.concat "; " (List.map value_string l)) ^ "]"
11  | Closure (_, _, _) -> "<closure>"
12  | Taintedval (_, v) -> "Tainted(" ^ (value_string v) ^ ")"
13  | SecretVal _ -> "<secret>"
```

Figure 5.14: *env_string* and *value_string* used by *exec*

5.2 Conclusion

The `interpreter.ml` file defines an interpreter for the functional language, ensuring secure execution through dynamic taint analysis and careful management of trust contexts. Each pattern in the `eval` function is designed to handle specific types of expressions securely and efficiently, propagating taint as necessary and raising errors when security violations are detected. This robust design guarantees that both trusted and untrusted code can be executed safely, maintaining the integrity of the execution environment.

CHAPTER 6

Tests

This chapter will introduce all the tests that we added to check that the requirements are met and also that the runtime check and limitations are met.

6.1 BinOp's test

As shown in *Figure 6.1* we have the following tests, which are basics tests:

- `test_addition` it checks that the binary operation of the sum is working;
- `test_multiplication` it checks that the binary operation of the multiplication is working;
- `test_boolean` it checks the creation of boolean values;
- `test_conditional` it checks that the interpreter can handle a condition;
- `test_concatenation` it check that the interpreter can concatenate two different strings;
- `test_subtraction` it tests the subtraction between two numbers.

```

1  let test_addition() =
2    let empty_env = empty_env in
3    let addition_expr = BinOp ("+", CstInt 74, CstInt 32) in
4    let addition_result = eval empty_env false addition_expr in
5    assert (addition_result = Int 106);;
6
7  let test_multiplication() =
8    let empty_env = empty_env in
9    let multiplication_expr = BinOp ("*", CstInt 18, CstInt 21) in
10   let multiplication_result = eval empty_env false multiplication_expr in
11   assert (multiplication_result = Int 378);;
12
13  let test_boolean() =
14    let empty_env = empty_env in
15    let bool_expr_true = CstBool true in
16    let bool_result_true = eval empty_env false bool_expr_true in
17    assert (bool_result_true = Bool true);;
18
19  let test_conditional() =
20    let empty_env = empty_env in
21    let conditional_expr = If (CstBool true, CstInt 10, CstInt 5) in
22    let conditional_result = eval empty_env false conditional_expr in
23    assert (conditional_result = Int 10);;
24
25  let test_concatenation() =
26    let empty_env = empty_env in
27    let concat_expr = BinOp ("+", CstStr "lb", CstStr "ts") in
28    let concat_result = eval empty_env false concat_expr in
29    assert (concat_result = String "lbts");;
30
31  let test_subtraction() =
32    let empty_env = empty_env in
33    let subtraction_expr = BinOp ("-", CstInt 3, CstInt 5) in
34    (try
35      let _ = eval empty_env false subtraction_expr in
36      assert false
37    with
38      | _ -> assert true);;

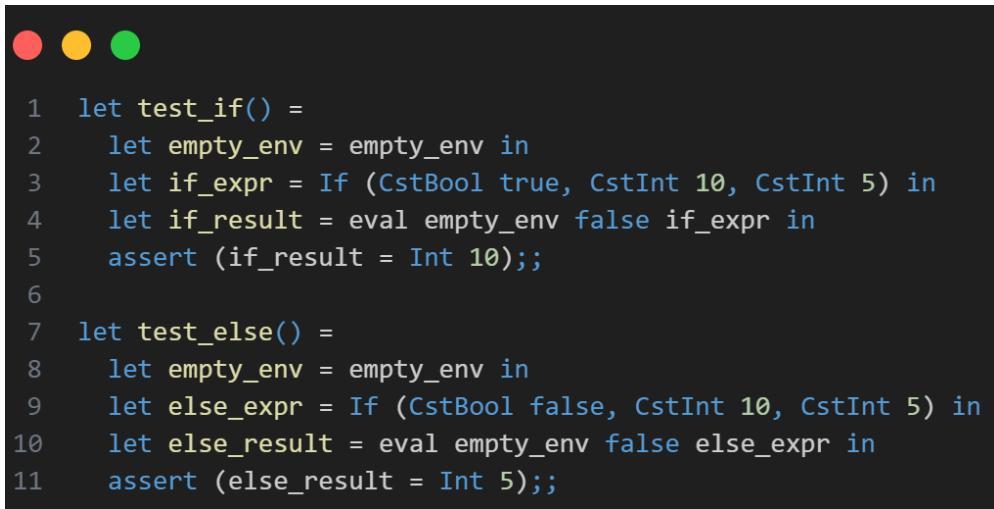
```

Figure 6.1: *binOps tests in main.ml*

6.2 If tests

As shown in *Figure 6.2* we have the following tests, which are basics tests:

- `test_if` it checks that the interpreter correctly execute the first expression of the if;
- `test_else` it checks that the interpreter correctly execute the second expression of the if.



```

1  let test_if() =
2    let empty_env = empty_env in
3    let if_expr = If (CstBool true, CstInt 10, CstInt 5) in
4    let if_result = eval empty_env false if_expr in
5    assert (if_result = Int 10);;
6
7  let test_else() =
8    let empty_env = empty_env in
9    let else_expr = If (CstBool false, CstInt 10, CstInt 5) in
10   let else_result = eval empty_env false else_expr in
11   assert (else_result = Int 5);;

```

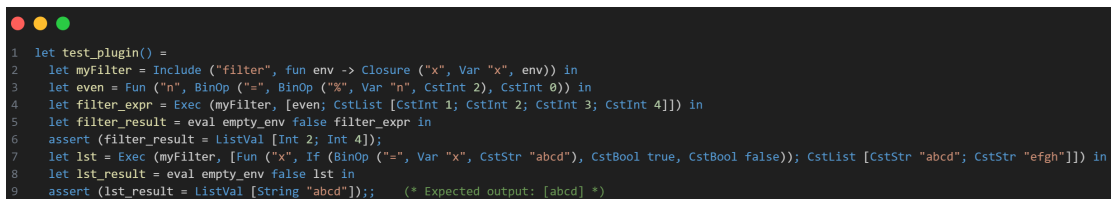
Figure 6.2: *If tests in main.ml*

6.3 plugin tests

As shown in *Figure 6.3* we have the following tests, which are basics test:

- `test_plugin` checks that we can include a plugin in a normal code.

The test that will check that we cannot include a plugin inside a trust block are done in the trust block section



```

1  let test_plugin() =
2    let myFilter = Include ("filter", fun env -> Closure ("x", Var "x", env)) in
3    let even = Fun ("n", BinOp ("=", BinOp ("%"), Var "n", CstInt 2), CstInt 0)) in
4    let filter_expr = Exec (myFilter, [even; CstList [CstInt 1; CstInt 2; CstInt 3; CstInt 4]]) in
5    let filter_result = eval empty_env false filter_expr in
6    assert (filter_result = ListVal [Int 2; Int 4]);
7    let lst = Exec (myFilter, [Fun ("x", If (BinOp ("=", Var "x", CstStr "abcd"), CstBool true, CstBool false)); CstList [CstStr "abcd"; CstStr "efgh"]]) in
8    let lst_result = eval empty_env false lst in
9    assert (lst_result = ListVal [String "abcd"]);; (* Expected output: [abcd] *)

```

Figure 6.3: *plugin test in main.ml*

6.4 Execute tests

As shown in *Figure 6.4* we have the following test:

- `test_execute` test that `exec` is blocked if we are inside a trust context from a trust block. The second part of the test check that `exec` can execute outside of a trusted context.

```

1 let test_execute() =
2   (* test that exec is blocked if we are inside a trust context from a trust block*)
3   let myFilter = Include ("filter", fun env -> Closure ("x", Var "x", env)) in
4   let even = Fun ("n", BinOp ("=", BinOp ("%"), Var "n", CstInt 2), CstInt 0)) in
5   let tainted_filter_expr = Exec (myFilter, [even; CstList [CstInt 1; CstInt 2; CstInt 3; CstInt 4]]) in
6
7   (* test that exec is blocked if we are inside a trusted context*)
8   (try
9     let _ = eval_empty_env true tainted_filter_expr in
10    assert false (* Should never reach here *))
11 with
12 | Failure msg -> assert (contains_substring msg "[ERROR]: Plugins are not allowed in trust context");
13
14 (* test if exec can execute outside of a trusted context*)
15 let tainted_lst = Exec (myFilter, [Fun ("x", If (BinOp ("=", Var "x", CstStr "abcd"), CstBool true, CstBool false)); CstList [CstStr "abcd"; CstStr "efgh"]]) in
16 let tainted_lst_result = eval_empty_env false tainted_lst in
17 assert(tainted_lst_result = ListVal [String "abcd"])

```

Figure 6.4: *execute tests in main.ml*

6.5 Trust block tests

As shown in the next 3 *Figures* we have the following test:

- `test_plugin_in_trustblock` test that we can use a function exposed by an handle with a plugin if it is a valid call;
- `test_plugin_in_trustblock2` test that check that we cannot import a plugin inside a function exposed by the handle of the trust block, even without executing the plugin;
- `test_nested_trust_block` test that we cannot create a trust block inside another trust block;
- `test_handle_within_exec` test that we cannot expose an exec call with an handle;
- `test_handle_outside_trust` test that we cannot create a handle outside a trust block.

```

1 let test_plugin_in_trustblock() =
2   let myFilter = Include ("filter", fun env -> Closure ("x", Var "x", env)) in
3   let even = Fun ("n", BinOp ("=", BinOp ("%"), Var "n", CstInt 2), CstInt 0)) in
4   let filter_expr = Exec (myFilter, [even; CstList [CstInt 1; CstInt 2; CstInt 3; CstInt 4]]) in
5   let filter_result = eval_empty_env false filter_expr in
6   assert (filter_result = ListVal [Int 2; Int 4]);
7
8 (*check that we cannot import a plugin inside a function exposed by the handle of the trust block, even without executing the plugin*)
9 let test_plugin_in_trustblock2() =
10   try
11     let myFilter = Include ("filter", fun env -> Closure ("x", Var "x", env)) in
12     let even = Fun ("n", myFilter) in
13     let handle_expr = Handle (even) in
14     let trust_block = Trust (CstInt 5, handle_expr) in
15     let _ = eval_empty_env false trust_block in
16     assert false; (* Should never reach here *)
17   with
18   | Failure msg -> assert (contains_substring msg "[ERROR]: Include block cannot be called inside a trust block");

```

Figure 6.5: *trust block in main.ml*

```

1 let test_nested_trust_block() =
2   try
3     let env = empty_env in
4     let nested_trust = Trust (CstInt 5, CstStr "nested trust") in
5     let trust_block = Trust (nested_trust, CstStr "outer trust") in
6     (* Evaluate the expression, but discard the result since we only care about the potential error *)
7     let _ = eval env false trust_block in
8     assert false; (* Should never reach here *)
9   with
10    | Failure msg -> assert (contains_substring msg "[ERROR]: Trust block cannot be declared inside a trust block");
11
12 let test_handle_within_exec() =
13   try
14     let env = empty_env in
15     let exec_call = Exec (Fun ("x", CstStr "exec dentro handle"), []) in
16     let inner_handle = Handle (exec_call) in
17     let _ = eval env true inner_handle in
18     assert false; (* Should never reach here *)
19   with
20    | Failure msg -> assert (contains_substring msg "[ERROR]: Handle block cannot contain an Exec call");

```

Figure 6.6: *trust block in main.ml*

```

1 let test_handle_outside_trust() =
2   try
3     let env = empty_env in
4     let handle_expr = Handle (CstStr "handle expression") in
5     let _ = eval env false handle_expr in
6     assert false; (* Should never reach here *)
7   with
8    | Failure msg -> assert (contains_substring msg "[ERROR]: Handle block cannot be called outside a trust block");

```

Figure 6.7: *trust block in main.ml*

6.6 Secrets Tests

As shown in *Figure 6.8* we have the following test:

- `test_trustblock_secret` test that the attempt to retrieve a secret with a plugin is not allowed.

```

1 let test_trustblock_secret() =
2   let secret = CstStr "abcd" in
3   let guess = Let ("guess", CstStr "abcd", Var "guess") in
4   let checkPwd = Fun ("guess", BinOp ("=", guess, secret)) in
5   let handle_expr = Handle (checkPwd) in
6   let trust_block = Trust (secret, handle_expr) in
7   let trust_result = eval empty_env false trust_block in
8   assert (trust_result = Bool true);
9
10  (*check that if we try to leak the secret of the trust block by using the plugin filter it fails*)
11  let myFilter = Include ("filter", fun env -> Closure ("x", Var "x", env)) in
12  let filter_expr = Exec (myFilter, [checkPwd; CstList [Secret "abcd"; CstStr "efgh"]]) in
13  (try
14    let _ = eval empty_env false filter_expr in
15    assert false
16  with
17    | Failure msg -> assert (contains_substring msg "[Blocked]: Prevention of secret data leakage"));

```

Figure 6.8: *secret test in main.ml*