University of Pisa

Academic year 2022/23
Course of Language-based technologies

Homeworks

Gianmarco Pastore

matr. 646236 – g.pastore11@studenti.unipi.it

Contents

1 Introduction

For the two homework I designed and built in OCAML a trusted execution environment of a functional language with linguistic primitives for:

- supporting enclave programming
- managing the secure execution of untrusted code
- enforcing confidentiality statically via a security type system for information flow
- enforcing integrity dynamically via dynamic taint analysis

I have decided to use some of the approaches seen in the lectures, together with some personal design choices, that I'm going to explain in the next chapters. If this report is not exhaustive enough, please check the code of the project which is well documented and contain more detailed information.

2 Interpreter

I have modeled the interpreter starting from the Abstract Syntax Tree, which is defined as following:

```
type exp =
  | EInt of int
  | EBool of bool
  | EString of string
  | EUnit of unit
  | Fun of (ide * exp typ) list * exp typ * exp
  | Den of ide
  | Op of ops * exp * exp
  | If of exp * exp * exp
  | Let of ide * exp typ * sec_level * exp * exp
  | Call of exp * exp list
  | Enclave of exp
  | End of ide * exp typ * sec_level * exp * exp
  | Access of exp * ide * exp typ
  | Include of exp
  | Execute of exp * exp typ
  | Declassify of exp
  | Endorse of exp
```

where `ops` are the possible operations, defined as:

```
type ops =
  | Sum  | Times  | Minus  | Div  | Mod  | Equal  |
    ↪ EqualStr  | Lesser  | Greater  | Lesseq  |
    ↪ Greateq  | Diff  | Or  | And
```

I have considered the following runtime values:

```
type runtime_value =
  | Int of int
  | Bool of bool
  | String of string
  | Unit of unit
  | Closure of
      (ide * exp typ) list * exp * runtime_value
        ↪ secure_runtime_value Env.env
  | REnclave of runtime_value secure_runtime_value Env.
    ↪ env
  | UCode of exp
```

As we can see, in addition to simple values (Int, Bool, String) we have:

- `Unit`: represent the unit type of Ocaml, has no particular use except for clarity in tests (similiar to a "skip" or "no-op")
- `Closure`: runtime value of functions, contains the data necessary for function execution
- `REnclave`: runtime value of enclaves, refer to 3 for further information
- `UCode`: runtime value for untrusted code, introduced with the operator `Include`

Finally, the environment is defined as:

```
type 'v env = (string * 'v) list
```

The following are the most noticeable aspects of the implementation:

- The expressions, that are directly transformed into runtime values, also contain a boolean attribute that correspond to the taint status (see 5).
- In the interpreter I have also decided to implement static type checking. Please, refer to the comments in the implementation for further information.
- When evaluated, function return a closure. Moreover, note that functions with an arbitrary number of arguments are supported. Anonymous functions are supported too.
- Before executing a program, it is typechecked statically with both the normal typechecker and the typechecker for information flow. During the execution is performed dynamic taint analysis.

In the source code (`bin/main.ml`, specifically), a wide variety of examples can be found, together with a short description and the expected result.

3 Enclave programming

I decided to consider the enclaves as a data structure that contains the data declared inside it. The actual OCAML implementation of an enclave consist of an environment of runtime values. It is like the one used by the interpreter,

but obviously it does not contain external values, because the enclave cannot see what's outside itslef.
To define the end of an enclave I designed the `End` operator, which works in the same way of a `Let`, but returns a `REnclave` runtime value.
The enclave can be accessed using the `Access` operator. This operator allows the access only to the fields of the enclave with security label Gateway (no direct access to Secret fields).

I report some of the checks that are enforced, regarding the usage of these operators (can be checked in the implemented tests):

- An `Enclave` must terminate with an `End`, otherwise `typechecker_error` is raised
- If trying to `Access` to something that is not an Enclave then `typechecker_error` ↪ is raised
- If trying to `Access` to a Secret field of an Enclave then `confidentiality_error` ↪ is raised
- If Enclave or `End` are used by tainted (untrusted) code then `runtime_error` ↪ is raised

The only case of improper usage not handled is the operator End outside Enclave. I assume that the programmer use it in a correct way, since untrusted codes cannot use it. I could have implemented a way to handle this case, e.g. with a boolean in the typecheker to indicate if we are in an enclave. But I decided that complicating the implementation just to handle this single case would not be worth the effort.

4 Confidentiality

Confidentiality has been enforced statically via a security type system for information flow. I used two different security levels for the lattice, Secret (High) and Gateway (Low), already introduced for the previous part on enclave programming.

```
type sec_level = Secret | Gateway
```

The security levels are assigned to variables when they are declared in `Let` ↪ statements. The type system certifies that the program satisfies non-interference.
The first thing that I implemented are some infix operators `++`,`<<=` to define, respectively, the $\sqcup$ and $\sqsubseteq$ operations for my lattice. The type system then has been implemented in a recursive way and with a static environment. It checks recursively the security levels in the code in order to find explicit and implicit information flows.
One design choice that I made is related to the handling of the security levels. In order to recursively typecheck the program I introduced some wrappers for

the security levels of more complex types (like functions, enclaves, untrusted codes).

```
type 'e sec_level' =
  | SL of sec_level
  | FSL of
      sec_level * (ide * exp typ) list * exp * 'e
        ↪ sec_level' Env.static_conf_env
  | ESL of sec_level * 'e sec_level' Env.static_conf_env
  | USL of sec_level * exp
```

This mechanism, also used in the normal typechecker, allow us to first evaluate the security level of the structure itself and then to analyze its body. Example: a function can be Gateway or Secret and its return value (result of the body evaluation) can be again (independently from the structure it belongs to) Gateway or Secret.
As already mentioned the type system can reject both explicit information flows in assignments and implicit flows in if and nested if clauses (thanks to the context variable). Unfortunately, as we have seen in the lectures, static type systems are more conservative so we must handle some false negatives, cases that satisfy non-interference but are rejected by the type system. In my implementation we can have some false negatives when a `If` is used. Since the type system is static we cannot know before runtime which branch of the if will be taken. This is a problem for my recursive implementation that in presence of an `If` must decide a resulting security level of the whole `If`. In order to be more clear I present the false negative example:

```
Let
  ( "test",
    TBool,
    Secret,
    EBool true,
    If (EBool true, EBool false, Den "test") )
```

This example satisfies non-interference but in the implementation the type system cannot decide **statically** if the whole `If` will be evaluated to a Secret value (var "test") or to a Gateway one (const false). So my implementation choice has been to reject every if with Gateway guard and with branches with different security level. It's a really conservative choice, but it's a price I had to pay for using a static mechanism.

The type system can also detect illegal flows in and through functions, i.e. Gateway functions cannot return Secret values. Let's consider the example reported in the first homework assignment:

```
Let
  ( "enc",
    TEnclave,
    Gateway,
```

```
    Enclave
      (Let
         ( "password",
           TString,
           Secret,
           EString "secret",
           End
             ( "y",
               TClosure,
               Gateway,
               Fun
                 ( [ ("guess", TString) ],
                   TBool,
                   Op (EqualStr, Den "guess", Den "
                      ↪ password") ),
               EInt 1 ) )),
    Call (Access (Den "enc", "y", TClosure), [ EString "
       ↪ secret" ]) )
```

This code tries to call a gateway function from the `Enclave` but it leaks a `Secret` password. It is correctly rejected by the type system. What if we want to allow this operation? Checking a password is a legitimate behaviour after all. For this purpose we can use the **Declassify** operator.

4.1 Declassify

This operator downgrades a secret (high) value into a public (low) value. The previous example using the Declassify is allowed by the type system.

```
Let
  ( "enc",
    TEnclave,
    Gateway,
    Enclave
      (Let
         ( "password",
           TString,
           Secret,
           EString "s3cr3t",
           End
             ( "y",
               TClosure,
               Gateway,
               Fun
                 ( [ ("guess", TString) ],
                   TBool,
                   Declassify
                     (Op (EqualStr, Den "guess", Den "
                        ↪ password")) ),
               EInt 1 ) )),
```

```
Call (Access (Den "enc", "y", TClosure), [ EString "
   ↪ s3cr3t" ]) )
```

Obviously the declassify operator cannot be used inside untrusted codes and moreover cannot be use on data that depends on tainted values. Otherwise an attacker could have tried to possibly trigger Declassification using tainted values.

5 Integrity and utrusted codes

Integrity has been enforced dinamically via dynamic taint analysis. First of all I modified the structure of runtime values to carry information about the taint status.

```
type 'v secure_runtime_value = 'v * bool
```

Taint status is a boolean: true for tainted data and false for untainted. For the rest of the design I considered these questions:

- How new taint is introduced? 5.1
- How taint propagates when execution progresses? 5.2
- How taint is checked during execution? 5.3

Let's start from the first one.

5.1 Untrusted code

In my implementation tainted data can be introduced with an untrusted code (`UCode`). This type of data can be included in the program using the `Include` operator and executed **only** with the `Execute` operator. Notice that untrusted codes have their own type and are handled as other data structures: `Include` and `Execute` are similar to `Fun` and `Call`.
Both the `UCode` structure and its content are considered tainted and are the only possible source of taintness.

5.2 Taint propagation

My interpreter works recursively and taint is propagated recursively through a boolean variable . As already said each runtime value has its taint status. Taint is propagated in the code in many ways:

- In assignments the taint status is stored in the program environment along with the values.
- In Ifs the taint is propagated from the guard to the branches.
- In operations the result is tainted if at least one of the operands is tainted.
- In function calls if the parameter passed to the function is tainted, it is such also inside it.

5.3 Taint detection

The most relevant case that the analysis can detect is when a tainted function (`Closure`) is called. The reason is that function calls modify the normal flow of execution of the program, so we do not want them to be controlled by a potential attacker.
Another case in which the program is terminated is when a `Declassify` operation is triggered by tainted data. In this way attackers cannot influence declassification.
The remaining cases are related to `UCode` trying to use relevant operators like `Enclave`, `Declassify`, `Endorse`.

5.4 Endorse

The `Endorse` operator transforms a tainted value into an untainted one. This is a mechanism for taint sanitization that can be used only by trusted codes.

6 Conclusions

The properties and the constraints described in this report can be verified in the code, which is well documented. In particular it is fundamental, to prove the results, to run the tests in the `bin/main.ml` file.