Language-Based Technology for Security

Trustee

Marco Antonio Corallo

Contents

Overview

Trustee is the extension of a simple, statically typed, functional language for supporting security primitives. These functionalities include

- Block of *trusted* code, with different qualified level of confidentiality and a mechanism to downgrade them;
- *Plugin*, that allows to include *untrusted* code;
- Static *Information-Flow* Analysis to prevent data leakage;
- *Dynamic Taint Analysis* to keep trace of the tainted value;
- Assertion primitives for testing both properties and taintness.

This report collects the requirements of these functionalities and the main design choices taken.

Technologies

Trustee has been developed using the OCaml ecosystem, including:

- `ocamllex` as lexer generator;
- `Menhir` as parser generator;
- `Dune` and `Makefile` as build system;
- `ppx_deriving`, `ppx_test` and `ppx_expect` for preprocessing and testing.

You can install the dependencies by `make setup` and then build the interpreter by `make`.

The Language

Trustee is strongly inspired by Ocaml, Java and Rust, is strongly and statically typed and uses an enriched type system that prevents and avoids in advance security errors and data leakages.
The easiest, fastest and most non-ambiguous way to describe the syntax of Trustee is commenting both the regular expressions for tokens and the structure of the grammar of the language.
However, this could be a bit verbose and it is not the main aim of the project, so I just list below some example to get in touch with the language.
If you are interested in the details, you can take a look to the files `lexer.mll` and `parser.mly`.

Syntax

Despite the structure of the language is strongly inspired by Ocaml, it present some syntactical differences and simplifications.
Unlike in OCaml, there are no free variables. So there is no *let* construct but only *let-in*:

```
1 let x = 5 in x
```

For (also recursive) functions there is a construct *let-fun* similar to the OCaml's *let-rec*:

```
1 let fun fact n =
2     if n = 0 then 1
3     else n * fact (n - 1)
4 in fact 5
```

However, in order to have a *static* type system, functions declaration make use of **mandatory** *type annotations*, so the factorial can be actually written as

```
1 let fun fact ( n : int ) : int =
```

```
2       if n = 0 then 1
3       else n * fact (n - 1)
4 in fact 5
```

Functions can also be used without declaration, this allows to have both *lambdas*

```
1 "anon-fun" |> lambda (s : string) : string -> " with
      annotation"
```

and recursive lambdas

```
1 5 |> (
2       fun fact ( n : int ) : int =
3           if n = 0 then 1
4           else n * fact (n - 1)
5 )
```

Type annotations are available in every other construct, but they are not mandatory.

Data Types

Trustee provides the most common data types: *integers*, *floats*, *chars*, *booleans* and *strings*.
Furthermore, it provides **homogeneous** *lists* of values and **heterogeneous** *tuples*.

Type	**Literal examples**	**Operators**	**Meaning**
int	`-5`, `0`, `42`	`+ - * / %`	Arithmetic op. on ints
float	`0.15`, `.0002`, `0.1e-22`	`+. -. *. /.`	Arithmetic op. on floats
string	`"Hello World"`	`^`	Concatenation of strings
boolean	`true`, `false`	`not && ||`	Logical op.
tuple	`('a', 0, "hi!")`, `(0,1)`	`proj t i`	Projection of the i-th element of t

list	`[2, 4, 6, 8]`, `[]`, `["Hello!"]`	`hd l`
		`tl l`
		`e::l`
			Get the first element of l
			Get l without the first element
			Add e in head of l

I/O

There are I/O directives for each data type.
The format for the type `T` is `get_T`/`print_T`.
IO functions are still expressions, and thus evaluation returns a value.
In particular, `print` functions evaluate to the special value `Unit`.

```
1 let fun fact(n : int) : int =
2   let _ = print_int n in    // sequencing
3   if n = 0 then
4     1
5   else
6     n * fact (n - 1)
7 in get_int () |> fact
```

Remarks

What has been shown so far is enough to write common programs. However, some remarks of design choice of the core of this language can be useful:

- An empty program is still a correct program (with `Unit` value).
- Multiple-arguments functions are internally converted in the corresponding *curried* functions.
- The language requires *type annotation* for function parameter and return type.
- The construct `Proj t i` takes a tuple and an integer **literal**.
  That's motivated by the static type checking.
- I/O primitives are *"native"* functions: closures are pre-loaded into the environment and defined by means of an AST node that can be instantied only by invokations to these functions.
- Contrarily, `Proj` on tuples and list operators are not functions. They are simple expressions, and then they cannot be used as higher-order functions or passed in pipe.
  Again, that's motivated by the static type checking and will be updated with the introduction of *generics*.

Trusted Blocks

The idea behind *trusted blocks* is of blocks of code that group together trusted code and data. They can be used to store confidential data and operations on them, as well as to mark a snippet of code as trusted, maybe because already verified.

The syntax chosen for trusted blocks follows the specification:

```
1 let trust pwd = {
2   let secret pass = "abcd" in   // confidential data
3   let fun checkpwd (guess : string) : bool =
4     declassify(pass = guess) in
5   handle: {checkpwd}
6 } in pwd.checkpwd "trythispass"
```

A trusted block can contain only definitions and a non-empty, mandatory, `handle` block that specifies which functions can be accessed from the external environment.

A trusted block is treated as an environment binding names to informations, that are:

- data type, qualifier and confidentiality level in phase of type-checking;
- value and integrity level during the evaluation.

Indeed, the type-checker ensures, among other things, that non-public (i.e. not handled) fields are not accessed and it is the responsible for the *information-flow analysis* that prevents data leakage.
All of these informations are then *erased* before the evaluation, in which only the integrity informations are retained.
The `secret` keyword is used to specify a field with an *high* confidentiality level, while `declassify` downgrades it. More details on confidentiality and information-flow analysis will be given in Chapter 5.

Plugin

Plugins represent the opposite idea of the trusted blocks: they are snippets of *untrusted* code, provided by third parties.
Plugins are implemented as (un)Trusted blocks: they share the same syntax with *tb*s and they are pretty similar:

```
1 let plugin filter = {
2   let fun string_f (predicate : string → bool) (l :
       string list) : string list =
3     if l = [] then []
4     else
5       if predicate (hd l) then (hd l)::(string_f
          predicate (tl l))
6       else (string_f predicate (tl l))
7   in handle: {string_f}
8 } in filter
```

Despite the nature of plugins is very different from that of trusted blocks, the choice to maintain the same syntax is due to a better usability: developers must not memorize more than one syntax for blocks. Furthermore, it allows to have both private data and public functions and it makes the implementation easy, taking advantage of code reuse. Of course, the confidentiality informations binded to code into plugins are different from that of code into trusted blocks, as well as integrity informations. Again, more details will be given in the next chapter.

A plugin can be included by means of the following syntax: `<"filename">`, with which the definition of a plugin is statically parsed and loaded from `plugin/filename`.
If `filename` is not in the `plugin` directory, or doesn't contain a plugin definition, then an exception is raised, in accordance with the principle of *least privilege*.

The Type System

The type system has been designed and developed incrementally, starting from the simple type-checking algorithm that given an expression maps it to its type, keeping a *type environment*.

The first enrichment is given by a serie of checks on the structure of blocks:

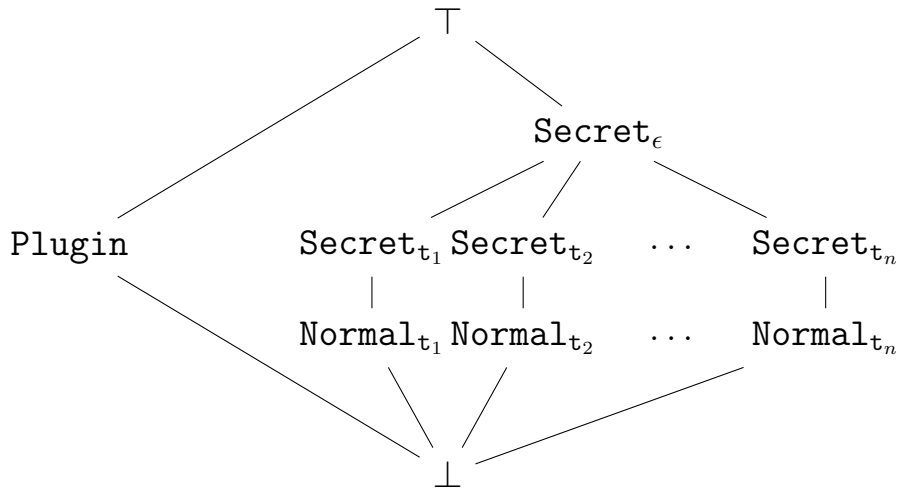
- deny nested blocks definition;
- enforce `secret` fields to be data and `handled` expressions to be (identifiers of) functions;
- deny the use of `secret` and `declassify` keywords outside of trusted blocks;
- only plugins can be included;

Subsequently, have been added the expected checks on the access to non-handled fields.

In the end, *information-flow* analysis has been implemented for preventing data leakage.
The *confidentiality* types are summarized in the following lattice, where we have:

- $\bot$ as starting confidentiality context;
- $Normal_{\texttt{tb}}$ as confidentiality level of non-secret code in a trusted block `tb`;
- $Secret_{\texttt{tb}}$ as confidentiality level of secret code in a trusted block `tb`;
- $Plugin$ as confidentiality level of plugins;

- $Secret_\epsilon$ as confidentiality level given by the interaction of more trusted blocks;
- $\top$ that indicates a possible data leak or malicious interaction.



This lattice has been designed so that:

- if `Normal` code interacts with `Secret` data of the same block, the former became `Secret`, propagating the higher confidentiality level;
- the interaction between two trusted blocks is lecit, as well as the interaction between two plugins with each other;
- the interaction between a trusted block and a plugin is recognized and blocked.

The choice of deny the direct interaction between trusted blocks and plugins may worsen the usability, since the type checker blocks also the interaction between plugins and **normal** code of trusted blocks, but it preserves the *soundness* of the IF analysis and thus the *safety* of the language.

Furthermore, this lattice automatically blocks the execution of plugins from inside trusted blocks, without the necessity to define an ad-hoc way to invoke plugins.

Ultimately, is not possible to statically distinguish code that does not *"use"* `secret` data from code that does via `declassify(.)`, since both are classified as `Normal`. Thus the safest way is to block it in

advance.

In order to improve the expressive power and the usability of the language, blocks have been designed to be *first-class citizens*.
A function can take a trusted block/plugin as argument or return it by specifying, in the corresponding type annotation, an *interface* that expose every handled function (type).

```
1 let plugin succ = {
2   let fun f (x : int) : int = x+1
3   in handle: {f}
4 } in
5 let fun apply (p : plugin{ f : (int → int) }) (x : int)
      : int = p.f x
6 in apply succ 5 // → 6
```

The same mechanism also works with trusted blocks, where the confidentiality level of each function in the interface is `Secret` by default and the user can modify it via the keyword `public`:

```
1 let trust t = {
2   let fun f : string = "this has conf. level Normal" in
      handle: {f}
3 } in
4 let fun f (b: trust{ public f: (unit → string)}) :
    string = b.f()
5 in f t
```

In conclusion, the static type system, together with the chosen confidentiality lattice, perform a conservative analysis that still leads to several benefits.
In addition to the well-known advantages of static typing, like early detection of type errors and *type erasure*, it allows to completely prevent data leakages and to easily satisfy the *Non-Interference* property.

The Interpreter

The enriched type system allows the interpreter to take into account only few things, taking advantage of *type erasure*. In particular, it computes the *Dynamic Taint Analysis* based on two levels of *integrity*: `Taint` and `Untaint`.
In this context the taint sources are given by input and plugin code, where the sinks are the trusted blocks.
Therefore, the analysis proceeds as expected by applying the well-known DTA semantics, and if at some point a trusted block is in a taint state, then the run-time raises a security exception.
In addition to that, a warning is printed if the evaluated program results in a taint state.

Run-time informations on taintness allowed to easily implement two types of assertions:

- classical assertions on boolean expressions:

```
1       let x = 5 in assert (x <> 5)
```

- and taintness-based assertions:

```
1       let x = 5 in assert taint (x <> 5)
```

  that evaluates the integrity state of the given expression, raising a failure in case of tainted value.

Finally, it is worth saying that blocks comparison is made by *reference* (or shallow) equality; this choice allows to avoid leaks by comparing a tb with a dictionary of different blocks.

Test cases

All the requirements specifications have been met in the development of *Trustee*, also the optional ones (i.e. the assertion mechanisms).

The testing process involved the three most common test types. In particular:

- each component of Trustee has been tested against a set of *unit* tests;
- *integration* tests has been incrementally applied on the overall system, increasing robustness and reliability;
- *functional* tests has been designed to cover the most common scenarios, considering also border values and malicious attempts.

Overall, more than 200 test cases has been designed and applied, including both automatic and manual, failing and succeed cases and different variations of the given `myFilter` case.

In conclusion, the language has been developed with a test-driven development model, that allowed the detection and the minimization of bugs. At the time this report is written everything seems to work as expected, although is well known that *"program testing can be used to show the presence of bugs, but never to show their absence"*.