Language-Based Technology for Security

Trusted-Fhree

Marco Antonio Corallo

Contents

Overview

Trusted-Fhree 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;
- *Plugin*, that is the possibility to include *untrusted* functions;
- 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

TFhree has been developed using the OCaml ecosystem, including:

- `ocamllex` as lexer generator;
- `Menhir` as parser generator;
- `Dune` and `Makefile` as build system.

Once installed the dependencies, you can easily `make` the system and run it specifying the source file name as argument.

The Language

The easiest, fastest and most non-ambiguous way to describe the syntax of TFhree 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

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

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

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

```
let fun fact n =
    if n = 0 then 1
    else n * fact (n - 1)
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 wrote as

```
let fun fact ( n : int ) : int =
    if n = 0 then 1
    else n * fact (n - 1)
in fact 5
```

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

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

and recursive lambdas

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

Type annotations are available in every other construct, but they are optional.

Data Types

TFhree 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 are evaluated to the special value `Unit`.

```
let fun fact(n : int) : int =
  let _ = print_int n in     // sequencing
  if n = 0 then
    1
  else
    n * fact (n - 1)
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, that is: closure 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 future introduction of *generics*.

Trusted Blocks

The idea of *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 is the following:

```
let trust pwd = {
  let secret pass = "abcd" in   // confidential data
  let fun checkpwd (guess : string) : bool =
    pass = guess in
  handle: {checkpwd}
} in // invokation
pwd.checkpwd "trythispass"
```

A trusted block can contain only definitions and a non-empty, mandatory, `handle` block that tells what functions of the block can be accessed from the external environment.
The evaluation of a trusted block led to a value that wraps an environment containing the names and values defined into the block.
The informations about the confidentiality level of these names are instead statically checked in phase of type-checking, then they are *erased*.
The type-checker also ensures that there are not nested block definitions, as well as for plugins.
Handled functions will be of `Public` *confidentiality type*, while *secret* vars will be of the $\texttt{secret}_{\texttt{tname}}$ one and the remaining definitions `Private`. Here, `tname` is the name of the trusted block inside which the variable is defined.
More details about the handling of confidentiality will be given in Chapter 5.

Plugin

The *Plugins* represent the opposite idea of the trusted blocks; indeed, they are snippets of *untrusted* code, provided by third parties.
The syntax chosen for plugins is the following:

```
let plugin filter = {
  let fun string_f (predicate : string → bool)
        (l : string list) : string list =
    if l = [] then []
    else
      if predicate (hd l) then
        (hd l)::(string_f predicate (tl l))
      else (string_f predicate (tl l))
  in string_f
} in // invokation
filter
    (lambda (x: string) : bool → (x="0"))
    ["0","1","2"]
```

Contrarily to trusted blocks, a plugin cannot contain more than one definition and, furthermore, it must be a function definition.
Indeed, plugins are designed for providing shared utilities and functions and they are nothing more than wrappers for functions.
This can be seen in the example, where the invokation of the plugin is done like it was a normal function.
Despite plugins are generally seen as a function, can be invoked and returned, they cannot be passed as argument to other functions. This enforcement seems to be a useless constraint, but ensures that a plugin cannot be invoked from inside a trusted block.
Trusted blocks and plugins are thus not considered as first-class citizen. A plugin should be just executed.
A plugin can be imported in a program by means of the `include <filename>` directive, that parses and loads the definition of the plugin from the file `plugin/filename`.

The Type System

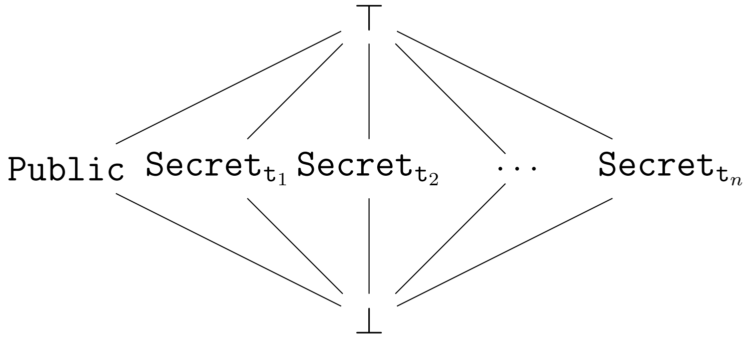
The type system has been designed and developed incrementally, starting from the simple type-checking algorithm that given an expression maps it to its type, keeping a *type environment*.

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

- deny nested blocks definition;
- enforce the type of the expression wrapped by a plugin to be a function;
- only data can be secret into trusted blocks;
- expressions into the `handle` block must be identifiers of functions defined before.

Subsequently, has been added the expected checks to the access to fields of trusted blocks.

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


$\top$
Public
$\texttt{Secret}_{\texttt{t}_1}$
$\texttt{Secret}_{\texttt{t}_2}$
$\cdots$
$\texttt{Secret}_{\texttt{t}_n}$
$\bot$

At first glance it may seems strange to see `Public` and `Secret` as *incomparable*, but I choose to use these labels to label respectively plugin code/handled functions and secret variables. In particular, secret variables of the block B_i will be typed as $\texttt{Secret}_i$.
At this point we have:

- $\perp$ as initial confidentiality context;
- `Public` for handled functions of trusted blocks and plugin code
- $\texttt{Secret}_i$ for secret variables of block `i`.

Then, the propagation goes on conservatively till the end of the type checking. If the confidentiality type returned by the type checker is $\top$, then the program could have a data leakage.

In conclusion, static typing allows to check the presence of data leakage, and prevent it, in advance of the execution.
Furthermore, it ensures that the program is well-typed, lightening the load of the interpreter.

The Interpreter

The enriched type system allows the interpreter to take into account only few things. 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.
Thus, the analysis goes on as expected applying the well-known DTA semantics. If the evaluated program is in a taint status, a warning is printed to the user.

In addition to that, the interpreter does a further check for preventing that plugin are passed as argument to function calls. Although it should be a type system task, it is simpler to check at run-time, bringing to an *hybrid* type checker.

In conclusion, two types of assertions are been implemented: classical assertions on boolean expressions:

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

and taintness-based assertions:

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

The latter evaluates the integrity status of the given expression, raising a failure in case of tainted value.

Test cases

All the requirements specifications have been met in the development of *TFhree*, also the *"optional"* ones (i.e. the assertion mechanisms).
Each component of TFhree has been tested on simple and more factitious test cases.
In particular, the type checker and the interpreter has been tested on a large battery of tests, including several variation of `myFilter`, tests that should succeed and tests that should fail.
The project is accompanied by a set of these test-cases and some plugins to test (and please, report) the new security primitives.

The language has been developed with a test-driven development model, that allowed the detection and the minimization of bugs and 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"*.