ObSec: Declassification policies using type abstraction

ObSec is a simple object-oriented language that supports type-based declassification. ObSec was introduced in the paper Type Abstraction for Relaxed Noninterference to be presented at ECOOP 2017 (http://conf.researchr.org/home/ecoop-2017)

This tutorial covers:

- how to install a web interpreter (ObSec Pad) for the ObSec language
- a basic presentation of the language syntax
- the type-based declassification examples that appears in the paper.

Installation instructions

We provide three ways to access to the ObSec Pad (although we strongly recommend the first one!):

1. **Using the online ObSec Pad** at https://pleiad.cl/obsec (https://pleiad.cl/obsec) .
2. **With the virtual machine**, which can be downloaded here (https://pleiad.cl/paper_53_ubuntu_vm_obsec.ova)
3. **Local execution of the ObSec Pad**, which can be downloaded here (https://pleiad.cl/_media/research/software/obsec/obsec-web.zip) .

The **online ObSec Pad** does not require any installation and is always up-to-date. If you want to get a snapshot of the current state of the interpreter (as of April 18, 2017), then follow the instructions for points 2 or 3.

The virtual machine

We have set up a virtual machine with an Ubuntu installation and Java 1.8. The virtual machine image was created with Virtual Box 5.1.18.

1. Install Virtual Box 5.1.18 or higher.
2. Import the Virtual Machine Image
3. Start the Virtual Machine
4. Use the following credentials to log in: username: dev , password: qwe123
5. Open a terminal. (Make sure you are at home.)
6. Run

```
./obsec-web/bin/obsec-web
```

This should start a server on port 9000. To open the ObSec Pad go to http://localhost:9000/obsec/ (http://localhost:9000/obsec/)

Local execution of the ObSec Pad

To run the ObSec Pad locally, all you need is Java 1.8. Also check that you have Java 1.8 in the path.

The following instructions are Unix valid commands. If you have Windows installed in your machine, you need to perform the corresponding commands.

1. Unzip the file.

   ```
   unzip obsec-web.zip
   ```

2. Go to the “obsec-web ” folder:

   ```
   cd obsec-web
   ```

3. Add run permissions to the binary file:

   ```
   chmod +x bin/obsec-web
   ```

4. And then run:

   ```
   ./bin/obsec-web
   ```

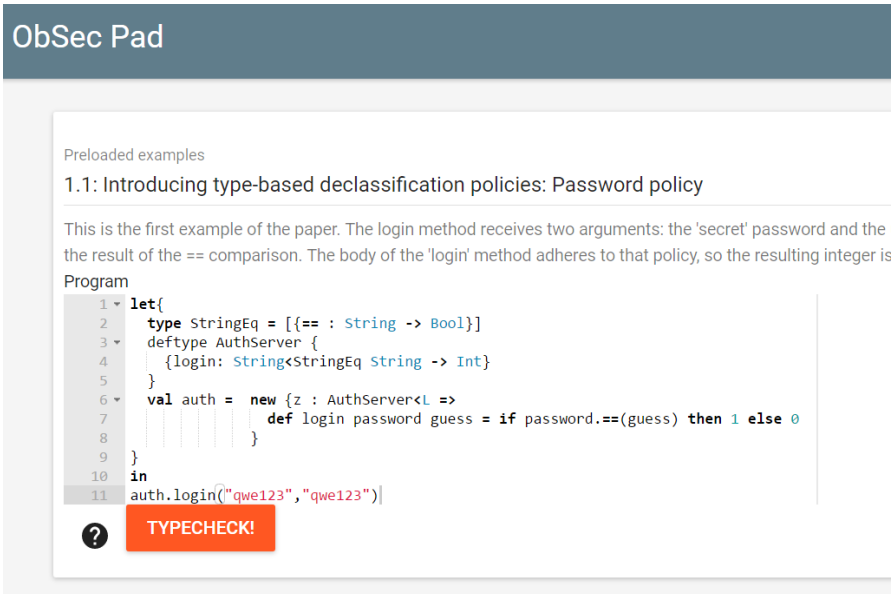
   This should start a server in the port 9000. To open the ObSec Pad, go to http://localhost:9000/obsec/ (http://localhost:9000/obsec/)

5. In case the port 9000 is already taken, run:

   ```
   ./bin/obsec-web -Dhttp.port=<available-port-number>
   ```

ObSec Pad: Funtionality

The following screenshot shows how the ObSec Pad looks like



Note that even though we provide some predefined examples, we suggest you finish this tutorial first. It will help you understand the syntax of ObSec and how to define and use type-based declassification policies.

The ObSec Pad has the minimal functionality you expect for an interpreter:

- Details of the language syntax.
- Type checking. It gives you the type of the program or an error if the program is not well-typed or if it has a syntax error.
- Execution. If the program was well-typed (as the result of clicking "TYPECHECK!"), the interface will show a button to execute it.

Introduction to ObSec : Warm up

ObSec is a minimal object-oriented language, which includes the following expressions: variables, object definitions, method invocations as well as built-in values. The implementation of ObSec also includes a number of convenient forms (not present in the formal core in the paper): integers, strings, boolean, string lists, as well as local variable definitions and type alias declarations.

Example of valid programs in ObSec are:

```
1.+(2)
```

```
"qwe123".length()
```

```
if "qwe123".length().==(6) then 1 else 0
```

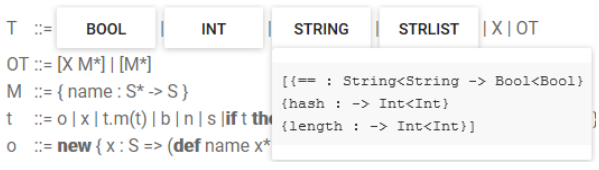
```
mklist("b","c","a")
```

```
let{
 val s = "qwe123"
 val l = s.length()
} in
if l.==(6) then 1 else 0
```

Types

The distinguishing feature of ObSec are security types: a security type `S` is a two-faceted type `T<U`, where `T` (resp. `U`) is the private (resp. public) facet. **However for the sake of presentation of the language syntax, the first examples will contain only types of the form `T<T`. To write such types (where the private and public facet are equals), you can use either the syntax `T<L` or just `T`.**

ObSec includes predefined types `Int`, `String`, `Bool`, and `StrList` lists of strings). You can see the type interface of built-in types in the Syntax Dialog in the ObSec Pad.



You can define and use object types in different ways. Below, we illustrate by creating an object type with a login method that takes two public strings and returns a public integer:

- Inline structural type definition: the type is defined (anonymously) where it is used

  ```
  [{login: String String -> Int}]
  ```

- Type alias declaration: gives a name (type alias) to an object type, convenient to reuse the same object type definition.

  ```
  type AuthServer  = [{login: String String -> Int}]
  ```

- Type definition: this is just a syntactic sugar for a type alias declaration, which additionally allows the definition of *recursive* object types, as discussed in the following paragraph.

  ```
  deftype AuthServer {
      {login: String String -> Int}
   }
  ```

Recursive types

The convenient way to define recursive types in the implementation of ObSec is using the `deftype` keyword. For example, the following code snippet defines the type of binary trees of strings. Methods `left` and `right` use the defined type `BinTree`.

```
deftype BinTree{
    {isEmpty: -> Bool}
    {value: -> String}
    {left: -> BinTree}
    {right: -> BinTree}
}
```

The above code is equivalent to:

```
type BinTree = [X
    {isEmpty: -> Bool}
    {value: -> String}
    {left: -> X}
    {right: -> X}
]
```

where the type variable `X` represents the defined recursive type. This core syntax is the one adopted in the paper, but which is less convenient to use than `deftype`.

Objects

ObSec does not have classes, instead one defines literal objects. An object is a collection of method definitions, where method names are unique. The object definition explicitly names the self variable (`z` in the example below) and binds it in method bodies. The self variable is ascribed with the type of the object: `AuthServer<L` indicating that the created object is public. The object below just has a method `login` that receives two arguments: `password` and `guess`. The type of the arguments are obtained from the `login` method signature in `AuthServer`.

```
new {z : AuthServer<L =>
  def login password guess = if password.==(guess) then 1 else 0
}
```

Executable example:

```
let {
    type AuthServer = [{login : String String -> Int}]
    val auth = new {z : AuthServer<L =>
                  def login password guess = if password.==(guess) then 1 else 0
                }
} in
auth.login("qwe123","qwe123")
```

Recursive methods

We now present an example with a recursive method definition. The following code shows how to use the self variable (`z`) to implement a recursive method `contains` over a list. The type of `listHelper` is defined in the type ascription of `z`:

```
let {
  val listHelper =  new {z : [{contains : StrList<L -> Bool<L}] <L =>
                      def contains myList  =
                       if myList.isEmpty()
                       then false
                       else
                         if myList.head().==("a")
```

```
                        then true
                        else z.contains(myList.tail())
                    }
}
in
listHelper.contains(mklist("b","c","a"))
```

Once you warm up with the examples, you can inspect the full syntax supported by the ObSec Pad. The syntax is essentially the one of the paper extended with built-in types and surface language constructions that makes programming more convenient.

Syntax

```
S       ::=     T < T                                                                                   (Security types)
T       ::=     Bool |Int |String | StrList | X | OT                                                    (Types)
OT      ::=     [X M*] | [M*]                                                                           (Object Type)
M       ::=     { name : S* -> S }                                                                      (Method signature)
t       ::=     o | x | t.m(t) | b | n | s |if t then t else t | mkList(t*) | let { TD* TA* VD* } in t  (Terms)
o       ::=     new { x : S => (def name x* = t)* }                                                     (Object)
TD      ::=     deftype{ M* }                                                                           (Type Declaration)
TA      ::=     type X = OT                                                                             (Type Alias)
VD      ::=     val x = t                                                                               (Value declaration)
b       ::=     true | false                                                                            (Booleans)
n       ::=     natural numbers                                                                         (Natural numbers)
s       ::=     string literals                                                                         (String literals)
X       ::=     identifier                                                                              (Type Variable)
```

You also can access the syntax definition in the ObSec Pad.

Security Typing in ObSec

The distinguishing feature of ObSec are security types `T<U`, where `T` (resp. `U`) is the private (resp. public) facet. For instance the type `String<H` (a shortcut to `String<[]`) represents to a private string, while `String<L` (a shortcut to `String<String` represents a public string. Note that for a security type `T<U` to be well-formed, T must be a subtype of U.

In other words, the empty interface type (the root of the subtyping hierarchy) denotes an object that hides all its attributes, which intuitively coincides with secret data, while the interface that coincides with the implementation type of an object exposes all of them, which coincides with public data.

The following example changes the type of the `password` argument in method `login` signature to `String<[]` making the `password` argument secret.

```
let{
    type AuthServer = [{login : String<[] String -> Int<L}]
    val auth = new {z : AuthServer<L =>
                def login password guess = if password.==(guess) then 1 else 0
              }
} in
auth.login("qwe123","qwe123")
```

This program is not well-typed because the implementation of the login method is using the `==` method, which is not available in the public facet. Hence the conditional expression `password.==(guess)` has type `Bool<H` and the whole `if` expression has type `Int<H`, which is not a subtype of `Int<L`. (As you can expect, `T1<U1` is a subtype of `T2<U2` if `T1` is a subtype of `U1` and `T2` is a subtype if `U2`). The ObSec type checker rejects this program to prevent an invalid information flow (from the secret password to a public result).

Declassification Policies in ObSec

As explained in the Sections “Introduction” and “Type-Based Declassification Policies” of the paper, in some scenarios, programs need to declassify *some* information about secrets as part of their functionality (as in the password checking algorithm).

Our main observation is that any interface between the empty type and the implementation type denotes declassification opportunities, and hence we can provide a *controlled* declassification approach.

For example we can just expose the `==` method for the `password` by including it in the public facet of the argument type.

```
let{
  type StringEq = [{== : String -> Bool}]
  type AuthServer = [{login : String<StringEq String -> Int<L}]
  val auth = new {z : AuthServer<L =>
              def login password guess = if password.==(guess) then 1 else 0
            }
} in
auth.login("qwe123","qwe123")
```

Now the program is well typed and we have a guarantee that only the `==` method of the `password` argument is invoked.

For instance, try to change the condition in the `if` to `password.hash().==(guess.hash())`. The resulting program is not well-typed, because `hash`

is not in the public facet of the `password` security type.

Progressive declassification policies

Declassification can be progressive, requiring several operations to be performed in order to obtain public data. For instance, “the result of comparing the hash of the password for equality can be made public”. This policy can be expressed with the type:

```
type StringHashEq = [{hash : -> Int<[{== : Int -> Bool}]}]
```

The implementation of the login method now adheres to the `password` policy and then the program is well typed.

```
let{
  type StringHashEq = [{hash : -> Int<[{== : Int -> Bool}]}]
  type AuthServer = [{login : String<StringHashEq String -> Int<L}]
  val auth = new {z : AuthServer<L =>
              def login password guess = if password.hash().==(guess.hash()) then 1 else 0
             }
} in
auth.login("qwe123","qwe123")
```

Variations to the above program that do not respect the policy and hence are not well-typed:

- Change the conditional expression to password.hash().+(1).==(guess.hash().+(1)).
- Change the conditional expression to password.length().==(guess.length()).

Recursive declassification

Recursive declassification policies are desirable to express interesting declassification of either inductive data structures or object interfaces (whose essence are recursive types). Consider for instance a secret list of strings, for which we want to allow traversal of the structure and comparison of its elements with a given string. This can be captured by the recursive type `StrEqList` defined as:

```
deftype StrEqList{
  {isEmpty: -> Bool<L}
  {head: -> String<StringEq}
  {tail: -> StrList<StrEqList}
}
```

To allow traversal, the declassification policy exposes the methods `isEmpty`, `head` and `tail`, with the specific constraints that a) accessing an element through `head` yields a `StringEq` (not a full `String`) and b) the `tail` method returns the tail of the list with the same declassification policy `StrEqList`.

The following program shows an implementation of a `contains` method over a list with the declassification policy `StrEqList`

```
let{
  type StringEq = [{== : String -> Bool}]
  deftype StrEqList{
    {isEmpty: -> Bool<L}
    {head: -> String<StringEq }
    {tail: -> StrList<StrEqList}
  }
  val listHelper =  new {z : [{contains : StrList<StrEqList -> Bool<L}]<L  =>
                      def contains myList  =
                        if myList.isEmpty()
                        then false
                        else
                          if myList.head().==("a")
                          then true
                          else z.contains(myList.tail())
                    }
}
in
listHelper.contains(mklist("b","c","a"))
```

Any method invocation over the `head` of the list (different than `==`) renders the program ill-typed. For instance, changing the inner `if` condition to `myList.head().hash().==(“a”.hash())`: the type checker reports “Both branches of an if expression must have the same type”. This is because the `else` branch of the outer `if` has type `Bool<H` while the `then` branch has type `Bool<L`.