Well-typed generic smart-fuzzing for APIs

Thomas Braibant[1,2], Jonathan Protzenko[2], and Gabriel Scherer[2]

[1] Cryptosense
thomas@cryptosense.com
[2] Inria
jonathan.protzenko@inria.fr, gabriel.scherer@inria.fr

1 Introduction

Despite recent advances in program certification, testing remains a widely-used component of the software development cycle. Various flavours of testing exist: popular ones include *unit testing*, which consists in manually crafting test cases for specific parts of the code base, as well as *QuickCheck-style* testing, where instances of a type are automatically generated to serve as test inputs. These methods of testing can be thought of as *internal* testing: the test routines need to access the internal representation of the data-structures that are used by the functions under test. They can also be thought of as *per-function* testing: a test suite is built (by hand, or automatically) for each function that must be tested.

We propose a new method of *external* testing that applies at the level of the *module interface*. The core of our work is a small embedded domain specific language to describe APIs, i.e., functions and data-types. Then, these API descriptions are used to drive the generation of test-cases. We have successfully used this method in two different contexts:

Test case generation. First, we implemented a library dubbed ArtiCheck that combines the functions exported by a given module interface to build elements of the various data-types exported by the module, and then checks that all the elements of these data-types meet user-defined invariants.

Smart fuzzing. Second, the first author reimplemented this methodology while working at Cryptosense to automate the analysis of (security) APIs. More precisely, Cryptosense's Testing library uses an API description to automatically exercise vendors' implementations of the said API.

2 The essence of external testing

In the present section, we illustrate the essential idea of external testing of APIs through a simple example in the context of OCaml. We consider a module `SIList` whose type `t` represents sorted lists of integers. This invariant is maintained by making `t` abstract: the user is forced to use functions from the interface to build instances of the type `t`.

```
module SIList: sig
  type t
  val empty: t
  val add: t -> int -> t
  val sorted: t -> bool
end
```

Given this signature, all a client can do is combine calls to `empty` and `add` to build new instances of `t`. We take the point of view that a client should not be able to violate type abstractions, e.g., building ill-formed applications like `add empty empty`. (This is obvious in the context of OCaml, but we also apply this rule to the security APIs we consider. However, in the context of security, well-typed does not necessarily mean *well-formed*) In essence, we want to represent well-typed applications of function symbols such as `empty` and `add`, and thus, we need to reify the types of these two functions as an OCaml data-type. This can be done using generalised abstract data-types (GADTs). For instance, we can define the type `('a,'b) fn` of *function signatures*. An element `('a,'b) fn` describes a function that has type `'a` and generates values of type `'b`.

```
type (_,_) fn =
| Ret: 'a ty -> ('a, 'a) fn
| Fun: 'a ty *  ('b, 'c) fn -> ('a -> 'b, 'c) fn
and 'a ty = {ident: string; mutable enum: 'a list}
let (@->) a b = Fun (a,b)
let ret a = Ret a
```

The type `'a ty` is used to keep track of the instances of `'a` that have been built. For types like `int`, we can populate the enumeration before-hand. For the abstract types like `t`, we can only populate this enumeration on the fly. Now, we can simply describe an API as an heterogeneous list of functions:

```
type signature = elem list
and  elem =
  Elem : string * ('a,'b) fn * 'a -> elem
let declare label fn f = Elem (label,fn,f)
```

The API of the `SIList` module can be encoded as follows.

```
let int_ty : int ty = ...
let t_ty : SIList.t ty = ...
let api =
  let open SIList in
  [ declare "empty" (ret t_ty) empty;
    declare "add"
    (int_ty @-> t_ty @-> ret t_ty) add]
```

Then, we exercise the API, applying the various functions it exports to suitable arguments. In effect, this may produce new suitable arguments that we can use to continue the test process; or, we may reach a fixpoint when the set of instances of types that we have cannot be used to produce new instances. (Of course, we can also ensure termination by putting a bound on the number of instances we want to produce.) What is important is that the process of exercising the functions of an API is closely related to the process of building instances of its types. In what follows, *testing* will denote one or the other.

3 Implementing a testing library

The simplistic design that is outlined in §2 has many shortcomings. Here are some hints about the improvements we implemented.

A better algebra of types. If the type of a function indicates that it returns a pair, we want to be able to break it apart, and use its components separately. The same goes for sums, we want to be able to discriminate between the two possible head constructors. Thus, we enrich the definition of `'a ty` to support sum, products, and atoms (arbitrary user-defined types that are used in an abstract manner). We also give ways to encode n-ary sums and records via two-ways mappings (e.g., we encode `'a option` as `() + 'a`).

Storing instances. Using lists to store instances of atomic types is inefficient, but there is no clear "one size fits all" solution. The right way to store instances of these types actually depends on the use-case. Sometimes, the user needs a set of the instances that may be quotiented by a suitable equivalence relation. Sometimes, keeping a sample of the instances that are created is sufficient. Sometimes, the user is only interested in enumerating a known set of values to bootstrap the creation of other instances. We propose various kinds of *containers* to store instances, and we make it possible for the user to choose the right one depending on the situation.

Iterating tests. The problem of testing functions over all possible inputs can nicely be presented as a kind of fixpoint computation. If we define the state of the iteration as a mapping from types to the set of their inhabitants, then an API can be considered as a simple state transformer. This presentation makes it possible to use an off-the-shelf fixpoint library, F. Pottier's `Fix` in ArtiCheck as a first approximation. However, using `Fix` does not scale up to the use-cases of Cryptosense's library. We solve this issue in two steps. First, we implement a library of *lazy enumerators* to store the set of test cases that need to be processed for each function. Then, we use an algorithm akin to formal derivatives to compute what is the extra work that should be done each time a value is added to a set of instances. Lazy enumerators and derivatives are crucial to mitigate the combinatorial explosion.

4 Use-cases

4.1 Examples with ArtiCheck

We used ArtiCheck on various examples: Red-Black trees, AVL trees, skew-heaps and BDDs. For instance, we checked that all the BDDs created by our library are reduced and ordered.

4.2 Examples at Cryptosense

We are using Cryptosense's library to reverse engineer (i.e., test) the behaviour of hardware devices that implement security APIs like PKCS #11.

5 Conclusion

This talk will expose the key abstractions that are used in the design of our two libraries and highlight the lessons learned.