DUCKEE GO: Dynamic and User-friendly ConcoliK Execution Engine in GO

Christopher Shao (cshao), Grace Yin (graceyin), Justin Restivo (jrestivo)

1 Overview

DUCKEE GO is a concolic execution engine for Go. It works by instrumenting the user's source code by converting it to an AST, and then modifying the AST. That is to say, replacing Go's native types with our concolic types. Then, as in 6.858's lab 3, our program concolically executes the instrumented code with various inputs, using the Z3 SMT solver to construct buggy inputs. We use the Go Z3 bindings provided by `aclements`.

DUCKEE GO is cross-platform, shown to work on Windows, MacOSX, Linux, and FreeBSD.

1.1 Concolic Execution

Concolic execution is a static analysis technique whose goal is to systematically explore all execution paths of a program. Starting from an initial input, a concolically executing program will generate constraints at each branch in the program. Then, after executing on that input, for each branch we encountered, the concolic execution engine will use an SMT solver like Z3 to generate inputs that would make the program branch the other way, and then repeat this process with these inputs. Eventually, the concolic executor will have tried many of the possible execution paths in the program, potentially uncovering bugs.

To do this, the program to be analyzed is modified (e.g. by augmenting the types in the program with constraints) so that operations will produce new values and constraints as the program executes.

1.2 Challenges in Go

There are several aspects of Go that make implementing concolic execution harder than in Python (as in lab 3).

- **Go is statically typed**
  To run things concolically, we need to create concolic types in order to hold their concrete values and symbolic constraints, and adjust operations to modify symbolic constraints as well. Since Go is statically typed, we have to indicate to use the concolic types everywhere the regular types appear. In particular, this affects function declarations and function calls, as will be explained in more detail in a later section.

- **Go is not Object Oriented**
  In Python, it is possible to subclass types like `int`, and then override their constructors and methods to also handle things symbolically. While interfaces do exist, we cannot subclass types as seamlessly in Go because interfaces only specify method headers, and don't allow for a list of state and super() methods as in python.

- **Go does not allow operator overloading**
  Go also prohibits us from overriding any of the arithmetic operators/comparators. Thus, using the arithmetic operators directly on concolic types will result in a type error. Soin addition to converting variables to their concolic types, we also need to change how we perform certain these arithmetic operations on our concolic types.

Our solution to these challenges is to produce a modified version of a program's source in order to our concolic types and operations.

2 METHODS

2.1 Concolic Execution

2.1.1 Concolic Types

We implemented concolic types for `bool`, `int`, and `map[int]int`. Each of these are structs comprised of a concrete value and a Z3 symbolic representation (`Bool`, `Int`, and `Array` from the `aclements` Z3 package). Operations on these concolic types update both the concrete and symbolic parts of the operation.

The following is a list of all the types and operations we support for concolic execution:

- ConcolicBool
  - `Not() -> ConcolicBool`
  - `Eq(ConcolicBool) -> ConcolicBool`
  - `And(ConcolicBool) -> ConcolicBool`
  - `Or(ConcolicBool) -> ConcolicBool`
- ConcolicInt
  - `Eq(ConcolicInt) -> ConcolicBool`
  - `NE(ConcolicInt) -> ConcolicBool`
  - `LT(ConcolicInt) -> ConcolicBool`
  - `LE(ConcolicInt) -> ConcolicBool`
  - `GT(ConcolicInt) -> ConcolicBool`
  - `GE(ConcolicInt) -> ConcolicBool`

  - `Add(ConcolicInt) -> ConcolicInt`
  - `Sub(ConcolicInt) -> ConcolicInt`
  - `Mul(ConcolicInt) -> ConcolicInt`
  - `Div(ConcolicInt) -> ConcolicInt`
  - `Mod(ConcolicInt) -> ConcolicInt`

  - `Not() -> ConcolicInt`
  - `And(ConcolicInt) -> ConcolicInt`
  - `Or(ConcolicInt) -> ConcolicInt`
  - `XOr(ConcolicInt) -> ConcolicInt`
  - `SHL(ConcolicInt) -> ConcolicInt`
  - `SHR(ConcolicInt) -> ConcolicInt`
  - `AndNot(ConcolicInt) -> ConcolicInt`
- ConcolicMap (only supports `map[int]int`)
  - `Get(ConcolicInt) -> ConcolicInt`
  - `Put(ConcolicInt, ConcolicInt)`

We attempted to implement ConcolicStrings as well. However, the Go Z3 bindings do not support strings. We looked at the other types supported by the Z3 package we use (implemented by Github user and Golang developer `aclements`), and considered using bit vectors to represent strings. However, we cannot compare equality between Z3 bit vectors of different lengths, which was a challenge we could not overcome. Additionally, none of the other Go Z3 bindings are as comprehensive as the `aclements` Z3 package.

2.1.2 Concolic Execution Engine

We essentially replicated the concolic execution functions from lab 3 in Go – `concolic_exec_input`, `concolic_force_branch`, `concolic_find_input`, `concolic_execs`. No significant modifications were made to their functionalities. Of note is that the concolic execution does not catch and address user-specified panics if they occur during a particular iteration, so the execution will stop if the program panics.

2.2 User Interaction

Users need to do several things in order to use DUCKEE GO.

- They need to specify which functions in which files should be instrumented to run concolically. This is done by providing the relevant file metadata about all the project source files that should be instrumented in a JSON file called `config.json` in the project directory. (A sample is provided in the directory `example`.)
- They should specify which variables are to be ”fuzzy” (i.e. which variables Z3 should solve its constraints for). There are functions `MakeFuzzyInt`, `MakeFuzzyBool`, and `MakeFuzzyMapIntInt` that each take as arguments a name for the Z3 variable and a value. During the instrumentation process, variables assigned to the results of these functions should be made fuzzy. Otherwise, variables are treated as constants. These functions are intended to have no effect on normal program execution, as they just return the second argument, but are there as markers for instrumenting.
  For example, the statement `a := 5` should be converted to in the original source `a := MakeFuzzyInt("a", 5)` if `a` is to be treated as fuzzy during concolic execution. The call to `MakeFuzzyInt` returns 5, and allows the original source to run normally.
- Finally, users should have a function named `main` that serves as the entry point to their program.

With these in mind, DUCKEE GO will build a instrumented copy of the entire project directory to `./tmp/DuckieConcolic/`, then creates new instrumented versions of the files specified in `config.json`. Finally, the user’s `main` function is copied to `[project_path]/tmp/DuckieConcolic/userMain.go`, and a new main function to start the concolic execution is added.

This functionality is provided by the script `src/run.sh`. The user, however, must provide the path of the JSON config file (which should be in the root package directory).

2.3 Code Instrumentation

To instrument the user’s source code, we use the Go `ast` and `astutil` packages. We convert the source file(s) into ASTs, which we then perform a dfs-like walk and modify the tree, adding instrumentation via `astutil`. Then, we then convert the AST back into a Go program to be run concolically.

2.3.1 *Instrumenting Types*

Every instance of an `int`, `bool`, or `map[int]int` is converted to its corresponding concolic type. Every instance of the supported operations is also converted to its corresponding function on concolic types.

Variables specified through the `MakeFuzzy[TYPE]()` functions (e.g. `x := MakeFuzzyInt("x", 5)` are converted to concolic types whose symbolic constraint are fresh variables that Z3 will solve for. Otherwise (e.g. `x := 5`), their symbolic constraints are constants with whatever value they are assigned to.

2.3.2 *Instrumenting Branches*

As in lab 3, during execution of a particular path, we keep track of the path constraints introduced by branches by appending to a global list of `z3.Bool`s, called `currPathConstrs`. The concolic execution engine then uses these constraints to construct and solve constraints to go down other execution paths.

Since every `bool` is converted to a concolic `bool`, we additionally instrument if statements to branch on the concrete values of the concolic `bool` representing their conditions. In the true branch, we add a statement that adds the symbolic constraint of the condition to `currPathConstrs`, and in the false branch, we add a statement that instead adds the negation of that constraint. Finally, for if statements with no else clause, we add an else clause so that we can add the negation of the constraint.

2.3.3 *Instrumenting Function Calls*

Programs we are testing can make calls both to functions whose source code we do augment and those we do not (e.g. they are not specified through the JSON config, or they are from external libraries). Ideally, we would like to pass concolic values as arguments to these calls, so that we can continue building constraints inside called functions, at least when they are functions we instrument. However, in the latter case, we cannot modify the signatures of these functions to operate on concolic values, so we cannot directly pass concolic values as arguments, as typechecking will fail. Go does not support inheritance, so we cannot subvert this problem by having `concolicInt` be a subclass of `int`, for example.

To handle function calls, we implemented a symbolic stack, much like a normal call stack, where we push and pop symbolic constraints. Suppose we are running an instrumented function `foo` that calls another function `bar`. `foo` is instrumented so that before it calls `bar`, `foo` pushes the symbolic constraints for `bar`'s arguments in reverse order. We do not instrument the signature of `bar`, and just directly pass it only the concrete values. We can instrument `bar` so that once it starts, it checks this stack and pops off the symbolic values of its arguments. Then, using the concrete values passed into it, `bar` reconstructs its arguments as concolic types, and then executes concolically. Once `bar` finishes, it pushes any symbolic constraint on the return value onto the stack, and returns the concrete value. Then `foo` can reconstruct the returned value as a concolic type by popping from the stack.

If `bar` is left uninstrumented, it can still execute properly only using the concrete values. Once `bar` finishes, `foo` can see that the symbolic constraints on the stack were not used and removes them. If the return value of `bar` can be converted to a concolic type, `foo` also converts it as if the return value were defined as a constant (without `MakeFuzzy[TYPE]`).

Of note is that DUCKEE GO only supports functions that return a single value.

2.3.4 *Instrumenting User Entry Point*

We replace the user's main method with one that executes the user's main method concolically. It executes via the `go/reflect` package to call the user's main method. This requires the user's main method to be (1) renamed and (2) called on a dummy object (the `reflect` API requires this), which we have declared in our replacement main file and called `Handler`.

3 Results

We implemented a simplified version of the original, buggy zoobar transfer function from lab 3 in Golang. This is our ”interesting” example function. Our goal was to have the concolic execution engine find the same bugs that we would have found from lab 3, and print the inputs that would cause the program to go down each path. This has been successfully implemented and included in example/main.go:transfer. Upon building this sample program (which may be done by executing src/run.sh, a bash script that builds a project with instrumented code in src/tmp based on a specified project path included in src/run.sh), then running it (go run src/tmp/main.go).

Upon executing the instrumented code in src/tmp, we will see a pretty printed list of input values that hit each branch. Each variable and its value is specified in the following format:

ITERATION iter_num:
[
input_variable $->$ value,
...
]

4 Avenues for Improvement

Some features we wish we added:

- **ConcolicString:** Support for concolic strings would be helpful to catch bugs like injection attacks. Unfortunately, as mentioned previously, the Go Z3 package we used doesn't support string operations, and implementation via bit strings is infeasible due to issues involving comparing bit vectors of different lengths.

- **ConcolicStruct:** Support for concolic structs would be helpful, as structs are commonly found in Go programs. To do so would require recursively access struct fields and adding constraints to each of them.

- **Improvements to ConcolicMap:** It may be possible to support ConcolicMaps of types other than `map[int]int`. However, this would probably require supporting other types (like strings) as well, since maps involving `bool`s don't seem very interesting.

- **Function Calls with multiple return values:** It is possible to support function calls with multiple return values by essentially replicating for return values the logic for pushing and popping arguments.

- **Support for panics:** It may be possible to concolically execute programs with user-defined panics by modifying `concolicExec`.

5 Conclusion

In this paper, we introduce and outline our concolic executor for Go. We have shown it to successfully generate inputs for all branch on some simple example programs.

After implementing this project, we now understand why a concolic executor for Go does not yet exist. There are many traits about Go and its relevant packages that make this inherently difficult. It's like C, but instead of having the acute control over everything that C has, Go sweeps many details under the rug. Yet, control of these key details are necessary to instrument the code. However, we believe that this is a good starting point for such a concolic execution engine in Go.