yinyang

Release v0.2

Dominik Winterer, Chengyu Zhang, Zhendong Su

Aug 18, 2021

CONTENTS

yinyang is a fuzzing framework for SMT solvers. It realizes three tools *typefuzz*, *opfuzz* and *yinyang*. Given a set of SMT-LIB seed formulas, each of the tools generates mutant formulas to stress-test SMT solvers. yinyang roughly operates in the following stages:

1. *Parsing:* First, yinyang parses a single or a set of SMT-LIB formulas to be used for fuzzing. yinyang's parser supports the SMT-LIB v2.6 standard and is customizable.
2. *Mutation:* Next, yinyang will mutate the parsed formula(s) using a mutation strategy. yinyang ships three mutation stragies. The most powerful one is generative type-aware operator mutation which mutates expressions within seed formulas and will by default generate 300 mutant formulas per parsed formula.
3. *Oracle Check:* Finally, yinyang will query the SMT solvers under test with the mutant formulas and compare the result against a test oracle. By default, such a test oracle would be a second SMT solver but it can also be fixed to be sat or unsat.

yinyang is intended for use by (1) SMT solver developers testing existing solvers, (2) researchers inventing new decision procedures to asses the robustness of their implementations, and (3) practitioners developing applications based on SMT solvers.

CHAPTER

ONE

INSTALLATION

To install a stable version of yinyang use:

` pip3 install yinyang `

The following commands clone yinyang and install the antlr4 python runtime.

```
$ git clone https://github.com/testsmt/yinyang.git
$ pip3 install antlr4-python3-runtime==4.9.2
```

CHAPTER

TWO

FUZZING SETUP

2.1 SMT-LIB seeds

To select SMT-LIB seed files for fuzzing SMT solvers with yinyang, edit `scripts/SMT-LIB-clone.sh` to select the logics for testing. Then use the following command to download the chosen benchmarks.

```
$ ./scripts/SMT-LIB-clone.sh
```

Alternatively, you can download the benchmarks directly from the website of SMT-LIB initiative or use your own benchmarks.

2.2 SMT solvers

To run typefuzz or opfuzz, you need to install two or more SMT solvers. The SMT-LIB initiative provides a comprehensive list of SMT solvers. Make sure that all SMT solver you consider for testing supports the chosen seeds.

If you can only use one SMT solver consider *Fusion*.

CHAPTER

THREE

BASIC USAGE

yinyang is a mutation-based fuzzer, i.e. it mutates a set of seed formulas using a mutation strategy and then uses the mutated formulas as the test seeds for SMT solvers. yinyang can so detect soundness bugs, invalid model bugs, crashes, segfaults, etc. With `typefuzz` we generate mutants by generating fresh expressions from the ones from the seed and root them by operators such as `=,distinct,+,-, *,/` by one another. You can run yinyang with the `typefuzz` strategy using the following command:

```
$ typefuzz "<solver_clis>" <seed_path>
```

- `<solver_clis>`: a sequence of SMT solvers command lines separated by semicolons. At least two SMT solvers command lines are necessary.
- `<seed_path>`: path to single seed or a directory containing the SMT-LIB seed files.

Example:

```
$ typefuzz "z3 model_validate=true;cvc4 --check-models -m -i -q" benchmarks
```

yinyang will by default randomly select formulas from the folder `./benchmarks`. By default SMT-LIB files larger than 20k will be ignored. yinyang will generate 300 mutants per seed formula and will run in an infinite loop. You can use the shortcut `CTRL+C` to terminate yinyang manually. If a bug has been found, the bug trigger is stored in `./bugs`.

Note: To catch invalid model bugs, you have to supply options to enable model validation in `<solver_clis>`. Also consider that you may need to supply options to enable model production and incremental mode to command lines in `<solver_clis>`.

Reducing a bug. After finding a bug, it is useful to produce a minimal test case before reporting the bug to save the SMT solver developers' time and effort. For many test cases, the C code reducer creduce does a great job. Besides, SMT-LIB specific reducer pydelta can be used.

CHAPTER

FOUR

CUSTOMIZATION

4.1 Options

yinyang provides the following options. Please consult `typefuzz --help` for a full list.

- `-i --iterations ITERATIONS` the number of iterations on each seed. (default: 300)
- `-m --modulo MODULO` specifies how often the mutants will be forwarded to the SMT solvers. For example, with 300 iterations and 2 as a modulo, 150 mutants per seed file will be passed to the SMT solvers. High modulo and iteration counts prioritize deeper mutations. (default: 2)
- `-t --timeout TIMEOUT` imposes a timeout limit (in seconds) on each SMT solver for solving mutant formula. (default: 8)
- `-d, --diagnose` forwards solver outputs to stdout e.g. for solver command line diagnosis.
- `-bugs BUGSFOLDER` (default: ./bugs)
- `-scratch SCRATCHFOLDER` specifies where the mutant formulas are temporarily stored. Note, if you run yinyang with several processes in parallel, each instance should have its own scratch folder. (default: ./scratch)
- `-km --keep-mutants` do not delete the mutants from the scratch folder. Warning: beware that this can quickly exhaust your entire disk space.
- `-q --quiet` do not output statistics and other output.
- `-fl", "--file-size-limit` file size limit on seed formula in bytes. (default: 20000)

4.2 Customize solvers configurations

If you want to test several SMT solver configurations at once the putting them as a commandline argument like `typefuzz "<solver_clis>" <seed_path>` may be inconvenient to you. Instead, you can modify the solver list in `.yinyang/Config.py`. The directory file need to be created by the user.

As an example consider:

```
solvers = [
    "z3 model_validate=true",
    "z3 model_validate=true smt.arith.solver=2",
    "z3 model_validate=true smt.arith.solver=3",
    "z3 model_validate=true smt.arith.solver=6",
    "cvc4 --check-models --produce-models --incremental --strings-exp -q",
]
```

You can then use `typefuzz "" <seed_path>` to run the above five solver configurations.

4.3 Customize bug detection

yinyang's bug detection logic is based on three lists: `crash_list, duplicate_list, ignore_list` of `.yinyang/Config.py` which you can customize. yinyang detects crash bugs by matching the stdout and stderr of the solvers in with the strings in the list``crash_list``. If yinyang detects a bug this way, it subsequently matches the crash message against all strings in `duplicate_list`. The `duplicate_list` is useful to filter out repeatedly occurring bugs from getting copied to `./bugs`. The `ignore_list` can be used to filter out errors occurring in a solver call. By default yinyang detects mutants returning non-zero exit codes as crashes except those that match with the `ignore_list`.

The below setup shows the three lists in `.yinyang/Config.py` that worked well in practice with Z3 and CVC4.

```
crash_list = [
    "Exception",
    "lang.AssertionError",
    "lang.Error",
    "runtime error",
    "LEAKED",
    "Leaked",
    "Segmentation fault",
    "segmentation fault",
    "segfault",
    "ASSERTION",
    "Assertion",
    "Fatal failure",
    "Internal error detected",
    "an invalid model was generated",
    "Failed to verify",
    "failed to verify",
    "ERROR: AddressSanitizer:",
    "invalid expression",
    "Aborted"
]

duplicate_list = [

]

ignore_list = [
   "(error ",
    "unexpected char",
    "failed to open file",
    "Expected result sat but got unsat",
    "Expected result unsat but got sat",
    "Parse Error",
    "Cannot get model",
    "Symbol 'str.to-re' not declared as a variable",
    "Symbol 'str.to.re' not declared as a variable",
    "Unimplemented code encountered",
]
```

CHAPTER

FIVE

FUSION

Fusion is a metamorphic testing approach than can work with a single SMT solver.If multiple suitable SMT solvers are available for your use-case, we recommend using `opfuzz` instead.

5.1 Basic Idea

The basic idea behind fusion is to fuse formula pairs into a new formula of known satisfiability (either both sat or both unsat). Given two seed formulas φ_1, φ_2 and variables x, y of φ_1 and φ_2 respectively, the idea is to

1. Concatenate the formulas φ_1 and φ_2
2. Add a fresh variable $z = f(x, y)$
3. Replace random occurrences of $x = g_x(y)$ and $y = g_y(x)$ within the concatenated formula

We call f a fusion function and g_x, g_y inversion functions.

5.2 Usage

```
$ python3 yinyang.py "<solver_clis>" -o <oracle> -s fusion <seed_path1> <seed_path2>
$ python3 yinyang.py "<solver_clis>" -o <oracle> -s fusion <seed_path>
```

where

- `<solver_clis>` a sequence of SMT solver commandlines separated by semicolons *;*. Note, since Fusion is a metamorphic testing approach, one SMT solver is sufficient.
- `<oracle>` desired test oracle result {sat, unsat}.
- `<seed_path1>, <seed_path2>` SMT-LIB v2.6 file of the same satisfiability, i.e. both either sat or unsat in accordance with the oracle.
- `<seed_path>` path to single seed or directory containing the SMT-LIB seed files, all of the same satisifiability.

Examples:

```
$ python3 yinyang.py "z3" -o sat -s fusion examples/phi1.smt2 examples/phi2.smt2
```

yinyang will test z3 by running fusion with 30 iterations on the two satisfiable seed formulas. The mutants generated yinyang will then be by construction satisfiable. In turn, with unsat as an oracle and two unsatisfiable seed formulas, fusion will generate unsatisfiable formulas.

```
$ python3 yinyang.py "z3" -o unsat -s fusion examples/phi3.smt2 examples/phi4.smt2
```

5.3 Seeds

Fusion requires the seeds that are pre-categorized to be either sat or unsat. Pre-categorized SMT-LIB scripts are available in the following repository. Fusion currently only supports non-incremental mode, e.g. LIA, LRA, NRA, QF_LIA, QF_LRA, QF_NRA, QF_SLIA, QF_S, etc. Fusion's applicability is constraint by the fusion function used.

5.4 Fusion functions

The configuration file `yinyang/config/fusion_functions.txt` specifies fusion and inversion functions. The format is the following:

```
#begin
<declaration of x>
<declaration of y>
<declaration of z>
[<declaration of c>]
<assert fusion function>
<assert inversion function>
<assert inversion function>
#end
```

Example:

The following code shows schematically fusion and inversion are described in `yinyang/config/fusion_functions.txt`.

```
#begin
(declare-const x Int)
(declare-const y Int)
(declare-const z Int)
(declare-const c Int)
(assert (= z (+ (+ x y) c)))
(assert (= x (- (- z y) c)))
(assert (= y (- (- z x) c)))
#end
```

The example realizes a fusion function for integer variables. First, the variables x,y,z are declared. Variable c will be substituted by a random but fixed integer constant. Then fusion function $z = f(x, y) = x + y + c$ is defined in the first assert block. Its corresponding inversion functions for x and y are described in the second and third asserts.

CHAPTER

SIX

BUILDING ON YINYANG

This section gives a brief overview of TypeFuzz's implementation and describes how researchers and practitioners can customize and extend TypeFuzz and yinyang.

6.1 Understanding TypeFuzz's implementation

The following file tree shows the most important files of typefuzz and includes a brief description.

```
yinyang
├── bin
│   └── typefuzz                          - main executable of typefuzz, cli interface
├── config
│   ├── Config.py                         - solver configurations, crash, duplicate,␣
↪ignore lists
│   └── typefuzz_config.txt               - typefuzz configuration file
├── src
│   ├── base                              - contains driver, argument parser, exitcodes,␣
↪etc.
│   ├── core
│   │   └──  Fuzzer.py                    - implements the fuzzing loop and the bug␣
↪checking oracle
│   ├── mutators
│   │   └── GenTypeAwareMutation
│   │       └── GenTypeAwareMutation.py - mutator integrating generative type-aware␣
↪mutations
│   └── parsing
│       ├── Ast.py                        - classes for scripts, commands, expressions,␣
↪etc.
│       ├── Parse.py                      - SMT-LIB v2.6 parser
│       ├── SMTLIBv2.g4                   - SMT-LIB v2.6 antlr4 grammar
│       └── Typechecker.py                - SMT-LIB v2.6 type checker
└── tests                                 - contains unit, integration, and regression␣
↪tests
```

When TypeFuzz is called from the command line, it executes *bin/typefuzz* containing the main function. After parsing the command line and reading in the seeds, the method *Fuzzer.py:run* is called. It randomly pops an SMT-LIB file from the seed list (*Fuzzer.py:142*), then parses (*Fuzzer.py:98*) and type-checks (*Fuzzer.py:146*) the SMT-LIB file. Next, we compute the set of unique expressions (*Fuzzer.py:148*) from the seed and pass it to a newly created mutator GenTypeAwareMutation (*Fuzzer.py:149*). The mutator is then called in a for-loop realizing n consecutive mutations (*Fuzzer.py:171*). Each mutated formula is then passed to the SMT solvers under test which checks for soundness bugs,

invalid model bugs, assertion violations, segfaults (*Fuzzer.py:185*) and dumps the bug triggers to the disk. For details on these checks, read the comments in the method *Fuzzer.py:test*.

Generative type-aware mutation's mutator class is realized in *GenTypeAwareMutation.py*. It takes a type-checked SMT-LIB script and the set of its unique expressions as arguments to the constructor. Then, we parse the configuration file (*yinyang/config/typefuzz_config.txt*) containing the operator signatures. The method *mutate* implements a mutation step. First, we call the method *get_all_subterms* to return a list of all expressions (*av_expr*) and another list with their types (*expr_type*). Next, we repeatedly choose a term t1 from the formula to be substituted by a term t2 (returned by *get_replacee*). If we could successfully construct such a term, then we substitute and return the mutated formula.

The *get_replacee(term)* method randomly chooses an operator from the list of candidate operators. The list of candidate operators contains all operators with a return type matching term's type and includes the identity operator *id*. Next, we pick a type-conforming expression from the set of unique expressions for every argument for the operator at hand and return the expression. The *get_replacee`method may fail, e.g., if we would have picked an operator of a conforming type but no term with conforming types to its arguments exist. To avoid this, we repeat the `get_replacee* method several times.

6.2 Customizing and extending yinyang

The yinyang framework has many tests to ensure the reliability of its mutators and the bug detection logic. All tests are integrated into a CI making sure that the bug-finding ability is preserved on every commit. yinyang adheres to the PEP 8 code quality standard. We briefly describe how researchers and practitioners can customize and extend the framework. For an in-depth overview of the yinyang framework, see the [documentation](https://yinyang.readthedocs.io/en/latest/).

6.3 Run TypeFuzz with other SMT Solvers

Besides Z3 and CVC4, TypeFuzz can be run with any other SMT solver such as [MathSAT](http://mathsat.fbk.eu), [Boolector](http://verify.inf.usi.ch/content/opensmt2), [Yices](http://yices.csl.sri.com/), and [SMT-Interpol](http://ultimate.informatik.uni-freiburg.de/smtinterpol/), etc. Since TypeFuzz is based on differential testing, it needs at least two solver configurations, ideally with a large overlap in the supported SMT logics. Furthermore, yinyang's type checker currently has stable support for string and arithmetic logics. Support for other logics is currently experimental but will be finalized shortly.

Solver configurations could either be specified in the command line or in the configuration file *yinyang/config/Config.py* such as: .. code-block:: text

solvers = ["yices-smt2 –incremental" "z3 model_validate=true", "z3 model_validate=true smt.arith.solver=6", "cvc4 –check-models –produce-models –incremental –strings-exp -q",

]

To run TypeFuzz with these four solver configurations in the config file, you would need to run *typefuzz "" <benchmark-dir>*. Note, the *crash_list* in yinyang/config/config.py, which may need to be updated ensuring that crashes by the new solver(s) are caught.

6.4 Devise a custom mutator

Fuzzing frameworks such as AFL and others have greatly benefited from the SE/PL community efforts to extend their mutation strategies. In the same spirit, we describe steps on how users can extend yinyang with custom mutators.

1. Add a new mutator class to *src/mutators*, e.g., *CustomGenerator.py*. A mutator takes a parsed SMT-LIB script as its input and returns the mutated script. The mutation should usually be implemented in a separate mutate method *CustomGenerator.py::mutate()*. For example, consider, *src/mutators/GenTypeAwareMutation/GenTypeAwareMutation.py* or *src/mutators/TypeAwareOpMutation.py*.
2. Provide an executable in the *bin* directory and add parser code to *base/ArgumentParser.py*.
3. Integrate the mutator in the fuzzing loop in *src/core/Fuzzer.py::run()*.

6.5 Extend the input language

Similar to many PLs, the [SMT-LIB language](https://smtlib.cs.uiowa.edu/language.shtml) is steadily augmented by new features, theories, etc. Furthermore, researchers use SMT-LIB dialects for their solver inputs (e.g. for sygus rewrite rules). To support such use cases, we have based yinyang's parser on an [ANTLR](https://www.antlr.org/) grammar that is simple to customize.

1. Extend grammar *src/parsing/SMTLIBv2.g4*.
2. Regenerate the grammar using *src/parsing/regenerate_grammar.sh*.
3. Extend parse tree visitor *src/parsing/AstVisitor.py* and AST implementation *src/parsing/Ast.py*.
4. If type checking is needed, augment the type checker in *src/parsing/Typechecker.py*.

6.6 Citing yinyang

The testing approaches implemented in yinyang are based on following two papers.

Type-Aware Operator Mutation (opfuzz) [pdf]

```
@article{winterer-zhang-su-oopsla2020
  author    = {Dominik Winterer and
               Chengyu Zhang and
               Zhendong Su},
  title     = {On the unusual effectiveness of type-aware operator mutations for
               testing {SMT} solvers},
  journal   = {Proc. {ACM} Program. Lang.},
  volume    = {4},
  number    = {{OOPSLA}},
  pages     = {193:1--193:25},
  year      = {2020},
}
```

Semantic Fusion (fusion) [pdf]

```
@inproceedings{winterer-zhang-su-pldi2020,
      title = {Validating SMT Solvers via Semantic Fusion},
      author = {Winterer, Dominik and Zhang, Chengyu and Su, Zhendong},
      year = {2020},
```

(continues on next page)

(continued from previous page)

```
      booktitle = {Proceedings of the 41st ACM SIGPLAN Conference on Programming
                   Language Design and Implementation},
      pages = {718-730}
}
```

6.7 Contact

We are always happy to receive your feedback or help you adjust yinyang to the needs of your custom solver, help you build on yinyang, etc. Reach out for us.

- Dominik Winterer - dominik.winterer@inf.ethz.ch
- Chengyu Zhang - dale.chengyu.zhang@gmail.com
- Jiwon Park - jiwon.park@polytechnique.edu
- Zhendong Su - zhendong.su@inf.ethz.ch