Fuzzinator Documentation

Release 18.3.1

Renata Hodovan, Akos Kiss

Mar 31, 2018

Quick Start

Random Testing Framework

CHAPTER 1

Introduction

Fuzzinator is a fuzzing framework that helps you to automate tasks usually needed during a fuzz session:

- run your favorite test generator and feed the test cases to the system-under-test,
- catch and save the unique issues,
- reduce the failing test cases,
- ease the reporting of issues in bug trackers (e.g., Bugzilla or GitHub),
- regularly update SUTs if needed, and
- schedule multiple SUTs and generators without overloading your workstation.

All the above features are fully customizable either by writing a simple config file or by implementing Python snippets to cover special needs. Check out some slides about *Fuzzinator* for a general overview, or see the Tutorial for a detailed walk-through on the config files.

To help tracking the progress of the fuzzing, *Fuzzinator* provides two interfaces:

- an interactive TUI (supported on Linux and Mac OS X) that gives a continuously updated overview about the currently running tasks, statistics about the efficacy of the test generators, and the found issues (and also supports reporting them); and
- a dump-mode (supported on every platform) that displays the news on line-based consoles.

Although *Fuzzinator* itself doesn't come with test generators (except for an example random character sequence generator), you can find a list of useful generators in the wiki.

1.1 Requirements

- Python >= 3.4
- pip and setuptools Python packages (the latter is automatically installed by pip)
- MongoDB (either local installation or access to remote database)

1.2 Install

The quick way:

```
pip install fuzzinator
```

Alternatively, by cloning the project and running setuptools:

```
python setup.py install
```

1.3 Usage

A common form of *Fuzzinator*'s usage:

```
fuzzinator --tui -U <path/to/the/config.ini>
```

1.4 Compatibility

Fuzzinator was tested on:

- Linux (Ubuntu 14.04 / 15.10 / 16.04)
- Mac OS X (OS X El Capitan - 10.11).

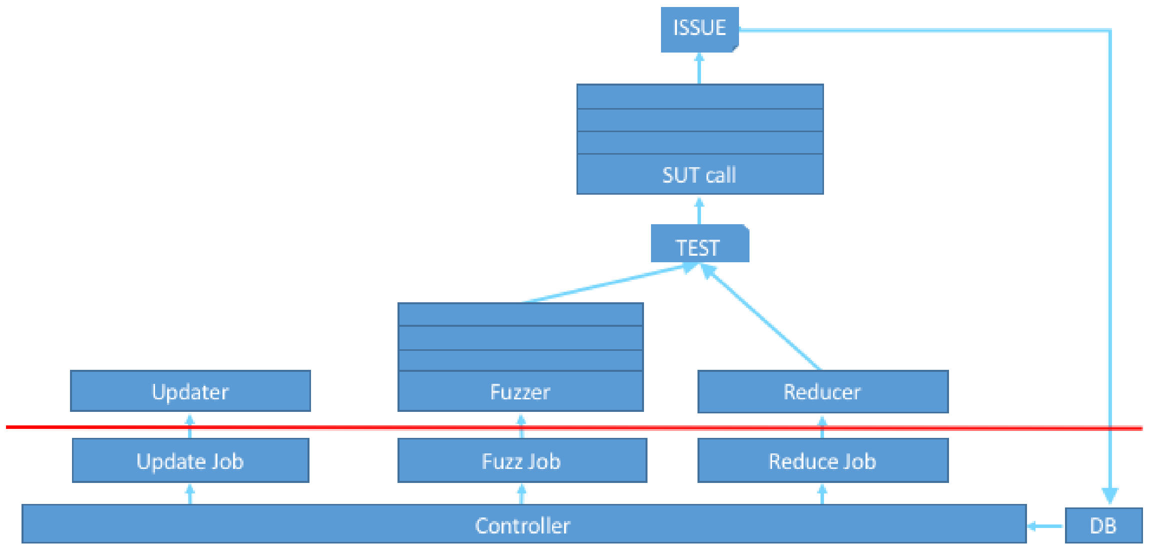
1.5 Acknowledgements

The authors are immensely grateful to Dr. Heinz Doofenshmirtz for the continuous inspiration.

CHAPTER 2

Tutorial

Fuzzinator is a framework helping you to deal with the common fuzzing tasks, like running fuzz jobs, updating the targets, and reducing the inputs that induced failures. The figure below shows a high-level overview of the components of the framework.



The red line represents an API boundary. Everything below it is part of the core infrastructure, while boxes above it are user-defined. However, you don't (necessarily) need to write a single line of code to describe your needs, since *Fuzzinator* can be configured through configuration *ini* files and it comes with several built-in building blocks that cover the most common scenarios and can be used out of the box.

In the next paragraphs, we will use the JerryScript project as our running example and we will incrementally build a configuration file to setup a fuzzing infrastructure for it. We will start from an absolute minimum configuration that will be extended step-by-step.

Although, the examples cover only a small subset of the provided building blocks, you can find the full list in the API Reference (under sub-packages with descriptive names). If none of them fits your needs, then you can still write your own snippet . . . or submit a feature request ;-)

2.1 Minimum Configuration

Let's start with the *minimum configuration* example that defines one SUT with `fuzzinator.call.StdinSubprocessCall`, expecting input from *stdin* and one test generator with `fuzzinator.fuzzer.RandomContent` that simply produces random strings.

```
# Sections starting with 'sut.' prefix define how target applications (a.k.a.,
# system-under-test or SUT) will be handled. The string after 'sut.' will be
# used as the identifier of the target. In this example, we deal with
# JerryScript.
[sut.jerry]
# StdinSubprocessCall will execute the target and return an issue dictionary if
# the target exits with a non-zero code.
call=fuzzinator.call.StdinSubprocessCall

# Define parameters expected by StdinSubprocessCall.
[sut.jerry.call]
# 'command' defines how SUT has to be executed.
command=./build/bin/jerry -
# Directory where 'command' has to be run.
cwd=</path/to/jerryscript/root/directory>

# Sections starting with 'fuzz.' prefix bind SUTs and test case generators.
[fuzz.jerry-with-random]
# Specify the SUT by referencing the appropriate config section.
sut=sut.jerry
# Specify the fuzzer by referring a Python callable.
fuzzer=fuzzinator.fuzzer.RandomContent
```

2.2 Fine Tuning SUT Calls

Now, if you would like to *fine-tune error detection* to do more than simply checking for a non-zero exit-code, then you can use two built-in solutions (or, again, you can implement your own version):

- `fuzzinator.call.ExitCodeFilter` for keeping issues only if the SUT exited with specific exit codes, and
- `fuzzinator.call.RegexFilter` for keeping issues only if the SUT printed messages on either *stdout* or *stderr* that matches some specific patterns.

We can extend the original example as follows:

```
[sut.jerry]
# ... define filters ...
# Properties named as 'call.decorate(N)' are Python decorators that can access
# the input & output of the wrapped methods (in this case, of
# StdinSubprocessCall) and can modify them. Here, they are used to filter the
# output issues. If decorators expect parameters, then they have to be defined
# in parameter sections named as 'sut.<SUT_NAME>.call.decorate(N)'.
```

```
call.decorate(0)=fuzzinator.call.ExitCodeFilter
call.decorate(1)=fuzzinator.call.RegexFilter

# Parameter section for ExitCodeFilter.
[sut.jerry.call.decorate(0)]
exit_codes=[132, 129]

# Parameter section for RegexFilter.
[sut.jerry.call.decorate(1)]
stderr=["(?P<msg>Assertion '.*' failed )at (?P<file>[^(]+)[(](?P<path>[^)]+)[)]:(?P
↪<line>[0-9]+)",
        "(?P<msg>Unreachable control path )at (?P<file>[^(]+)[(](?P<path>[^)]+)[)]:(?P
↪<line>[0-9]+)"]
```

However, issues not only can be filtered but also *extended with arbitrary information* that helps describing the circumstances of the failure. This extension can also happen with the above shown decorator approach. The next example shows how platform, git version, and ID information can be added using:

- *fuzzinator.call.PlatformInfoDecorator* adds an extra `'platform'` field to the issue dictionary, filled with OS information,
- *fuzzinator.call.SubprocessPropertyDecorator* adds a user-defined field with the output of a user-defined script, and
- *fuzzinator.call.UniqueIdDecorator* combines existing fields into an ID to help detect whether an issue is unique or a duplicate of an already known one.

```
[sut.jerry]
# .. extend issue with platform information ..
call.decorate(2)=fuzzinator.call.PlatformInfoDecorator
# .. extend issue with user-defined information ..
call.decorate(3)=fuzzinator.call.SubprocessPropertyDecorator
# .. add an id to the issue ..
call.decorate(4)=fuzzinator.call.UniqueIdDecorator

# *No* parameter section for the 2nd decorator as it needs none.

# Parameter section for the 3rd decorator.
[sut.jerry.call.decorate(3)]
# .. extend issue dictionary with a version field ..
property=version
# .. the value of version field is filled with the output of the next command ..
command=git rev-parse --short HEAD
# .. directory where 'command' has to be run (no need to copy the value of 'cwd'
# from the 'sut.jerry.call' section verbatim, extended interpolation syntax can
# help to reuse options) ..
cwd=${sut.jerry.call:cwd}

# Parameter section for the 4th decorator.
[sut.jerry.call.decorate(4)]
# .. compose the new id field from the msg and path fields previously found by
# RegexFilter ..
properties=["msg", "path"]
```

2.3 Updating SUTs and Reducing Tests

Similarly to the above, we can have control over SUT update and test reduce jobs as well. The following final example uses built-in building blocks again:

- *fuzzinator.update.TimestampUpdateCondition* for triggering the update based on the last modification time of the target binary,
- *fuzzinator.update.SubprocessUpdate* for updating the target via a script, and
- *fuzzinator.reduce.Picire* for reducing the size of test cases with Picire.

```
[sut.jerry]
# ... define update ...
update_condition=fuzzinator.update.TimestampUpdateCondition
update=fuzzinator.update.SubprocessUpdate
# ... define reduction ...
reduce=fuzzinator.reduce.Picire

# Parameter section for fuzzinator.update.TimestampUpdateCondition.
[sut.jerry.update_condition]
# Update SUT in every 12 hours.
age=12:00:00
path=${sut.jerry.call:cwd}/build/bin/jerry

# Parameter section for fuzzinator.update.SubprocessUpdate.
[sut.jerry.update]
# Script to execute to update.
command=git pull origin master &&
        ./tools/build.py --debug --clean
# Directory where 'command' has to be run.
cwd=${sut.jerry.call:cwd}
```

2.4 Etc...

There is more, e.g.:

- SUTs can take their input from files instead of *stdin*.
- Reducers are highly parametrizable.
- Test reduce jobs can deviate from fuzz jobs in the way their SUT is called.
- Fuzzers can be decorated the same way as SUT calls.
- Etc...

More complex configuration files are available in the `examples/configs` directory of the project (e.g., for WebKit, too).

CHAPTER 3

Fuzzinator Core: package `fuzzinator`

3.1 class `Controller`

class `fuzzinator.`**`Controller`**(*config*)

Fuzzinator's main controller that orchestrates a fuzz session by scheduling all related activities (e.g., keeps SUTs up-to-date, runs fuzzers and feeds test cases to SUTs, or minimizes failure inducing test cases) . All configuration options of the framework must be encapsulated in a `configparser.ConfigParser` object.

The following config sections and options are recognized:

- Section `fuzzinator`: Global settings of the framework.
  - Option `work_dir`: Work directory for temporary files. (Optional, default: `~/.fuzzinator`)
  - Option `db_uri`: URI to a MongoDB database to store found issues and execution statistics. (Optional, default: `mongodb://localhost/fuzzinator`)
  - Option `cost_budget`: (Optional, default: number of cpus)
- Sections `sut.NAME`: Definitions of a SUT named *NAME*
  - Option `call`: Fully qualified name of a python callable that must accept a `test` keyword argument representing the input to the SUT and must return a dictionary object if the input triggered an issue in the SUT, or `None` otherwise. The returned issue dictionary (if any) *should* contain an `'id'` field that equals for issues that are not considered unique. (Mandatory)

    See package *`fuzzinator.call`* for potential callables.
  - Option `cost`: (Optional, default: 1)
  - Option `reduce`: Fully qualified name of a python callable that must accept `issue`, `sut_call`, `sut_call_kwargs`, `listener`, `ident`, `work_dir` keyword arguments representing an issue to be reduced (and various other potentially needed objects), and must return a tuple consisting of a reduced test case for the issue (or `None` if the issue's current test case could not be reduced) and a (potentially empty) list of new issues that were discovered during test case reduction (if any). (Optional, no reduction for this SUT if option is missing.)

    See package *`fuzzinator.reduce`* for potential callables.

- Option `reduce_call`: Fully qualified name of a python callable that acts as the SUT's `call` option during test case reduction. (Optional, default: the value of option `call`)

  See package *`fuzzinator.call`* for potential callables.

- Option `reduce_cost`: (Optional, default: the value of option `cost`)

- Option `update_condition`: Fully qualified name of a python callable that must return `True` if and only if the SUT should be updated. (Optional, SUT is never updated if option is missing.)

  See package *`fuzzinator.update`* for potential callables.

- Option `update`: Fully qualified name of a python callable that should perform the update of the SUT. (Optional, SUT is never updated if option is missing.)

  See package *`fuzzinator.update`* for potential callables.

• Sections `fuzz.NAME`: Definitions of a fuzz job named *NAME*

- Option `sut`: Name of the SUT section that describes the subject of this fuzz job. (Mandatory)

- Option `fuzzer`: Fully qualified name of a python callable that must accept and `index` keyword argument representing a running counter in the fuzz job and must return a test input (or `None`, which signals that the fuzzer is "exhausted" and cannot generate more test cases in this fuzz job). The semantics of the generated test input is not restricted by the framework, it is up to the configuration to ensure that the SUT of the fuzz job can deal with the tests generated by the fuzzer of the fuzz job. (Mandatory)

  See package *`fuzzinator.fuzzer`* for potential callables.

- Option `batch`: Number of times the fuzzer is requested to generate a new test and the SUT is called with it. (Optional, default: 1)

- Option `instances`: Number of instances of this fuzz job allowed to run in parallel. (Optional, default: `inf`)

• Callable options can be implemented as functions or classes with `__call__` method (the latter are instantiated first to get a callable object). Both constructor calls (if any) and the "real" calls can be given keyword arguments. These arguments have to be specified in sections `(sut|fuzz).NAME.OPT[.init]` with appropriate names (where the `.init` sections stand for the constructor arguments).

• All callables can be decorated according to python semantics. The decorators must be callable classes themselves and have to be specified in options `OPT.decorate(N)` with fully qualified name. Multiple decorators can be applied to a callable `OPT`, their order is specified by an integer index in parentheses. Keyword arguments to be passed to the decorators have to be listed in sections `(sut|fuzz).NAME.OPT.decorate(N)`.

  See packages *`fuzzinator.call`* and *`fuzzinator.fuzzer`* for potential decorators.

Parameters **`config`** (*`configparser.ConfigParser`*) – the configuration options of the fuzz session.

Variables **`listener`** (`fuzzinator.ListenerManager`) – a listener manager object that is called on various events during the fuzz session.

`run`(**, max_cycles=None*)

Start the fuzz session.

Parameters **`max_cycles`** (*`int`*) – maximum number to iterate through the fuzz jobs defined in the configuration (defaults to `inf`).

3.2 class `EmailListener`

class `fuzzinator.`**`EmailListener`**(*event*, *param_name*, *from_address*, *to_address*, *subject*, *content*, *smtp_host*, *smtp_port*)

EventListener subclass that can be used to send e-mail notification about various events.

Parameters

- **`event`** – The name of the event to send notification about.
- **`param_name`** – The name of the event's parameter containing the information to send.
- **`from_address`** – E-mail address to send notifications from.
- **`to_address`** – Target e-mail address to send the notification to.
- **`subject`** – Subject of the e-mail (it may contain placeholders, that will be filled by parameter information).
- **`content`** – Content of the e-mail (it may contain placeholders, that will be filled by parameter information).
- **`smtp_host`** – Host of the smtp server to send e-mails from.
- **`smtp_port`** – Port of the smtp server to send e-mails from.

`send_mail`(*data*)

Send e-mail with the provided data.

Parameters **`data`** – Information to fill subject and content fields with.

3.3 class `EventListener`

class `fuzzinator.`**`EventListener`**

A no-op base class for listeners that can get notified by *`fuzzinator.Controller`* on various events of a fuzz sessions.

Note: Subclasses should be aware that some notification methods may be called from subprocesses.

`activate_job`(*ident*)

Invoked when a previously instantiated job is activated (started).

Parameters **`ident`** (*`int`*) – unique identifier of the activated job.

`invalid_issue`(*issue*)

Invoked when an issue seems invalid.

Parameters **`issue`** (*`dict`*) – the issue object that did not pass re-validation (listener is free to decide how to react, an option is to remove the issue from the database).

`job_progress`(*ident*, *progress*)

Invoked when an activated job makes progress.

Parameters

- **`ident`** (*`int`*) – unique identifier of the progressing job.
- **`progress`** (*`int`*) – for fuzz jobs, this is the number of already generated tests (number between 0 and the job's batch size); for reduce jobs, this is the current size of the test case being reduced (number between the original test size and 0).

`new_fuzz_job`(*ident*, *fuzzer*, *sut*, *cost*, *batch*)
Invoked when a new (still inactive) fuzz job is instantiated.

Parameters

- **`ident`** (*`int`*) – a unique identifier of the new fuzz job.
- **`fuzzer`** (*`str`*) – short name of the new fuzz job (name of the corresponding config section without the "fuzz." prefix).
- **`sut`** (*`str`*) – short name of the SUT of the new fuzz job (name of the corresponding config section without the "sut." prefix).
- **`cost`** (*`int`*) – cost associated with the new fuzz job.
- **`batch`** (*`int, float`*) – batch size of the new fuzz job, i.e., number of test cases requested from the fuzzer (may be `inf`).

`new_issue`(*issue*)
Invoked when a new issue is found.

Parameters **`issue`** (*`dict`*) – the issue that was found (all relevant information - e.g., the SUT that reported the issue, the test case that triggered the issue, the fuzzer that generated the test case, the ID of the issue - is stored in appropriate properties of the issue).

`new_reduce_job`(*ident*, *sut*, *cost*, *issue_id*, *size*)
Invoked when a new (still inactive) reduce job is instantiated.

Parameters

- **`ident`** (*`int`*) – a unique identifier of the new reduce job.
- **`sut`** (*`str`*) – short name of the SUT used in the new reduce job (name of the corresponding config section without the "sut." prefix).
- **`cost`** (*`int`*) – cost associated with the new reduce job.
- **`issue_id`** (*`Any`*) – `'id'` property of the issue to be reduced.
- **`size`** (*`int`*) – size of the test case associated with the issue to be reduced.

`new_update_job`(*ident*, *sut*)
Invoked when a new (still inactive) update job is instantiated.

Parameters

- **`ident`** (*`int`*) – a unique identifier of the new update job.
- **`sut`** (*`str`*) – short name of the SUT to be updated (name of the corresponding config section without the "sut." prefix).

`remove_job`(*ident*)
Invoked when an active job has finished.

Parameters **`ident`** (*`int`*) – unique identifier of the finished job.

`update_fuzz_stat`()
Invoked when statistics about fuzzers, SUTs, and issues (e.g., execution counts, crash counts, unique issue counts) are updated in the framework's database.

`update_issue`(*issue*)
Invoked when the status of an issue changed.

Parameters **`issue`** (*`dict`*) – the issue object that has changed.

update_load(*load*)

Invoked when the framework's load changes.

Parameters **load** (*int*) – number between 0 and controller's capacity.

warning(*msg*)

Invoked on unexpected events.

Parameters **msg** (*str*) – a string representation of the problem.

3.4 class ListenerManager

class fuzzinator.**ListenerManager**(*listeners=None*)

Class that registers listeners to various events and executes all of them when the event has triggered.

Parameters **listeners** – List of listener objects.

add(*listener*)

Register a new listener in the manager.

Parameters **listener** – The new listener to register.

CHAPTER 4

SUT Calls: package `fuzzinator.call`

4.1 class `AnonymizeDecorator`

class `fuzzinator.call.`**`AnonymizeDecorator`**(**args*, ***kwargs*)

Decorator for SUT calls to anonymize issue properties.

Mandatory parameter of the decorator:

- `old_text`: text to replace in issue properties.

Optional parameters of the decorator:

- `new_text`: text to replace 'old_text' with (empty string by default).
- `properties`: array of properties to anonymize (anonymize all properties by default).

Example configuration snippet:

```
[sut.foo]
call=fuzzinator.call.StdinSubprocessCall
call.decorate(0)=fuzzinator.call.AnonymizeDecorator

[sut.foo.call]
command=/home/alice/foo/bin/foo -

[sut.foo.call.decorate(0)]
old_text=/home/alice/foo
new_text=FOO_ROOT
properties=["stdout", "stderr"]
```

4.2 class `ExitCodeFilter`

class `fuzzinator.call.`**`ExitCodeFilter`**(**args*, ***kwargs*)

Decorator filter for SUT calls that return issues with `'exit_code'` property.

Mandatory parameter of the decorator:

- `exit_codes`: if `issue['exit_code']` is not in the array of `exit_codes`, the issue is filtered out.

The issues that are not filtered out are not changed in any way.

Example configuration snippet:

```
[sut.foo]
call=fuzzinator.call.StdinSubprocessCall
call.decorate(0)=fuzzinator.call.ExitCodeFilter

[sut.foo.call]
command=/home/alice/foo/bin/foo -

[sut.foo.call.decorate(0)]
exit_codes=[139]
```

4.3 class `FileReaderDecorator`

class `fuzzinator.call.`**`FileReaderDecorator`**`(`**args*, ***kwargs*`)`

Decorator for SUTs that take input as a file path: saves the content of the failing test case.

Moreover, the issue (if any) is also extended with the new `'filename'` property containing the name of the test case (as received in the `test` argument).

Example configuration snippet:

```
[sut.foo]
call=fuzzinator.call.SubprocessCall
call.decorate(0)=fuzzinator.call.FileReaderDecorator

[sut.foo.call]
# assuming that foo takes one file as input specified on command line
command=/home/alice/foo/bin/foo {test}
```

4.4 class `FileWriterDecorator`

class `fuzzinator.call.`**`FileWriterDecorator`**`(`**args*, ***kwargs*`)`

Decorator for SUTs that take input from a file: writes the test input to a temporary file and replaces the test input with the name of that file.

Mandatory parameter of the decorator:

- `filename`: path pattern for the temporary file, which may contain the substring `{uid}` as a placeholder for a unique string (replaced by the decorator).

The issue returned by the decorated SUT (if any) is extended with the new `'filename'` property containing the name of the generated file (although the file itself is removed).

Example configuration snippet:

```
[sut.foo]
call=fuzzinator.call.SubprocessCall
call.decorate(0)=fuzzionator.call.FileWriterDecorator
```

```
[sut.foo.call]
# assuming that foo takes one file as input specified on command line
command=/home/alice/foo/bin/foo {test}

[sut.foo.call.decorate(0)]
filename=${fuzzinator:work_dir}/test-{uid}.txt
```

4.5 class `GdbBacktraceDecorator`

class `fuzzinator.call.`**`GdbBacktraceDecorator`**`(`**args*, ***kwargs*`)`

Decorator for subprocess-based SUT calls with file input to extend issues with `'backtrace'` property.

Mandatory parameter of the decorator:

- `command`: string to pass to GDB as a command to run (all occurrences of `{test}` in the string are replaced by the actual name of the test file).

Optional parameters of the decorator:

- `cwd`: if not `None`, change working directory before GDB/command invocation.
- `env`: if not `None`, a dictionary of variable names-values to update the environment with.

The new `'backtrace'` issue property will contain the result of GDB's `bt` command after the halt of the SUT.

Example configuration snippet:

```
[sut.foo]
call=fuzzinator.call.SubprocessCall
call.decorate(0)=fuzzinator.call.GdbBacktraceDecorator

[sut.foo.call]
# assuming that {test} is something that can be interpreted by foo as
# command line argument
command=./bin/foo {test}
cwd=/home/alice/foo
env={"BAR": "1"}

[sut.foo.call.decorate(0)]
command=${sut.foo.call:command}
cwd=${sut.foo.call:cwd}
env={"BAR": "1", "BAZ": "1"}
```

4.6 class `LldbBacktraceDecorator`

class `fuzzinator.call.`**`LldbBacktraceDecorator`**`(`**args*, ***kwargs*`)`

Decorator for subprocess-based SUT calls with file input to extend issues with `'backtrace'` property.

Mandatory parameter of the decorator:

- `command`: string to pass to Lldb as a command to run (all occurrences of `{test}` in the string are replaced by the actual name of the test file).

Optional parameters of the decorator:

- `cwd`: if not `None`, change working directory before Lldb/command invocation.

- `env`: if not `None`, a dictionary of variable names-values to update the environment with.
- `timeout`: timeout (in seconds) to wait between two lldb commands (integer number, 1 by default).

The new `'backtrace'` issue property will contain the result of Lldb's `bt` command after the halt of the SUT.

Example configuration snippet:

```
[sut.foo]
call=fuzzinator.call.SubprocessCall
call.decorate(0)=fuzzinator.call.LldbBacktraceDecorator

[sut.foo.call]
# assuming that {test} is something that can be interpreted by foo as
# command line argument
command=./bin/foo {test}
cwd=/home/alice/foo
env={"BAR": "1"}

[sut.foo.call.decorate(0)]
command=${sut.foo.call:command}
cwd=${sut.foo.call:cwd}
env={"BAR": "1", "BAZ": "1"}
```

4.7 class `PlatformInfoDecorator`

class `fuzzinator.call.`**`PlatformInfoDecorator`**(**args*, ***kwargs*)

Decorator for SUT calls to extend issues with `'platform'` property.

The new `'platform'` issue property will contain the result of Python's `platform.platform`.

Example configuration snippet:

```
[sut.foo]
#call=...
call.decorate(0)=fuzzinator.call.PlatformInfoDecorator
```

4.8 class `RegexFilter`

class `fuzzinator.call.`**`RegexFilter`**(**args*, ***kwargs*)

Decorator filter for SUT calls to recognise patterns in the returned issue dictionaries.

Optional parameters of the decorator:

- key: array of patterns to match against `issue[key]` (note that 'key' can be arbitrary, and multiple different keys can be given to the decorator).

If none of the patterns matches on any of the fields, the issue is filtered out. The issues that are not filtered out are extended with keys-values from the named groups of the matching regex pattern.

Example configuration snippet:

```
[sut.foo]
call=fuzzinator.call.StdinSubprocessCall
call.decorate(0)=fuzzinator.call.RegexFilter
```

```
[sut.foo.call]
command=/home/alice/foo/bin/foo -

[sut.foo.call.decorate(0)]
stderr=["(?P<file>[^:]+):(?P<line>[0-9]+): (?P<func>[^:]+): (?P<msg>
↪Assertion `.*' failed)"]
backtrace=["#[0-9]+ +0x[0-9a-f]+ in (?P<path>[^ ]+) .*? at (?P<file>[^
↪:]+):(?P<line>[0-9]+)"]
```

4.9 function `StdinSubprocessCall`

`fuzzinator.call.`**`StdinSubprocessCall`**(*command*, *cwd=None*, *env=None*, *no_exit_code=None*, *test=None*, *timeout=None*, ***kwargs*)

Subprocess invocation-based call of a SUT that takes a test input on its stdin stream.

Mandatory parameter of the SUT call:

- `command`: string to pass to the child shell as a command to run.

Optional parameters of the SUT call:

- `cwd`: if not `None`, change working directory before the command invocation.
- `env`: if not `None`, a dictionary of variable names-values to update the environment with.
- `no_exit_code`: makes possible to force issue creation regardless of the exit code.
- `timeout`: run subprocess with timeout.

Result of the SUT call:

- If the child process exits with 0 exit code, no issue is returned.
- Otherwise, an issue with `'exit_code'`, `'stdout'`, and `'stderr'` properties is returned.

Example configuration snippet:

```
[sut.foo]
call=fuzzinator.call.StdinSubprocessCall

[sut.foo.call]
command=./bin/foo -
cwd=/home/alice/foo
env={"BAR": "1"}
```

4.10 class `StreamMonitoredSubprocessCall`

`class` `fuzzinator.call.`**`StreamMonitoredSubprocessCall`**(*command*, *cwd=None*, *env=None*, *end_patterns=None*, *timeout=None*, ***kwargs*)

Note: Not available on platforms without fcntl support (e.g., Windows).

4.11 function `SubprocessCall`

`fuzzinator.call.`**`SubprocessCall`**(*command*, *cwd=None*, *env=None*, *no_exit_code=None*, *test=None*, *timeout=None*, ***kwargs*)

Subprocess invocation-based call of a SUT that takes test input on its command line. (See *`fuzzinator.call.FileWriterDecorator`* for SUTs that take input from a file.)

Mandatory parameter of the SUT call:

- `command`: string to pass to the child shell as a command to run (all occurrences of `{test}` in the string are replaced by the actual test input).

Optional parameters of the SUT call:

- `cwd`: if not `None`, change working directory before the command invocation.
- `env`: if not `None`, a dictionary of variable names-values to update the environment with.
- `no_exit_code`: makes possible to force issue creation regardless of the exit code.
- `timeout`: run subprocess with timeout.

Result of the SUT call:

- If the child process exits with 0 exit code, no issue is returned.
- Otherwise, an issue with `'exit_code'`, `'stdout'`, and `'stderr'` properties is returned.

Example configuration snippet:

```
[sut.foo]
call=fuzzinator.call.SubprocessCall

[sut.foo.call]
# assuming that {test} is something that can be interpreted by foo as
# command line argument
command=./bin/foo {test}
cwd=/home/alice/foo
env={"BAR": "1"}
```

4.12 class `SubprocessPropertyDecorator`

`class` `fuzzinator.call.`**`SubprocessPropertyDecorator`**(**args*, ***kwargs*)

Decorator for SUT calls to extend issues with an arbitrary property where the value is the output of a shell subprocess.

Mandatory parameters of the decorator:

- `property`: name of the property to extend the issue with.
- `command`: string to pass to the child shell as a command to run.

Optional parameters of the decorator:

- `cwd`: if not `None`, change working directory before the command invocation.
- `env`: if not `None`, a dictionary of variable names-values to update the environment with.
- `timeout`: run subprocess with timeout.

Example configuration snippet:

```
[sut.foo]
call=fuzzinator.call.StdinSubprocessCall
call.decorate(0)=fuzzinator.call.SubprocessPropertyDecorator

[sut.foo.call]
command=./bin/foo -
cwd=/home/alice/foo

[sut.foo.call.decorate(0)]
property=version
command=git rev-parse --short HEAD
cwd=${sut.foo.call:cwd}
env={"GIT_FLUSH": "1"}
```

4.13 class `TestRunnerSubprocessCall`

class `fuzzinator.call.`**`TestRunnerSubprocessCall`**(*command*, *cwd=None*, *env=None*, *end_texts=None*, *init_wait=None*, *timeout_per_test=None*, ***kwargs*)

Note: Not available on platforms without fcntl support (e.g., Windows).

4.14 class `UniqueIdDecorator`

class `fuzzinator.call.`**`UniqueIdDecorator`**(**args*, ***kwargs*)

Decorator for SUT calls to extend issues with `'id'` property.

Mandatory parameter of the decorator:

- `properties`: array of issue property names, which are concatenated (separated by a space) to form the new `'id'` property.

Example configuration snippet:

```
[sut.foo]
call=fuzzinator.call.StdinSubprocessCall
call.decorate(0)=fuzzinator.call.RegexFilter
call.decorate(1)=fuzzinator.call.UniqueIdDecorator

[sut.foo.call]
command=/home/alice/foo/bin/foo -

[sut.foo.call.decorate(0)]
stderr=[": (?P<file>[^:]+):(?P<line>[0-9]+): (?P<func>[^:]+): (?P<msg>
↪Assertion `.*' failed)"]

[sut.foo.call.decorate(1)]
properties=["msg", "file", "func"]
```

CHAPTER 5

Fuzzers: package `fuzzinator.fuzzer`

5.1 class `AFLRunner`

class `fuzzinator.fuzzer.`**`AFLRunner`**(*afl_fuzz*, *input*, *output*, *sut_command*, *cwd=None*, *env=None*, *timeout=None*, *dictionary=None*, *master_name=None*, *slave_name=None*, ***kwargs*)

Wrapper around AFL to be executed continuously in a subprocess. The findings of AFL are periodically checked and any new test cases are returned as test inputs to the SUT. (Thus, all AFL findings are processed, extended, and filtered by any and all SUT decorators, uniqueness is determined, etc.)

For AFL, it is best not to run multiple instances in parallel.

Mandatory parameters of the fuzzer:

- `afl_fuzz`: path to the AFL fuzzer tool.
- `sut_command`: the string to append to the command string used to invoke AFL, probably the same string that is used for *`fuzzinator.call.SubprocessCall`*'s command parameter (the `{test}` substring is automatically replaced with the `@@` input file placeholder used by AFL).
- `input`: the directory of initial test cases for AFL.
- `output`: the directory that will store the findings of AFL (all occurrences of `{uid}` in the string are replaced by an identifier unique to this fuzz job).

Optional parameters of the fuzzer:

- `cwd`: if not `None`, change working directory before invoking AFL.
- `env`: if not `None`, a dictionary of variable names-values to update the environment with (`AFL_NO_UI=1` will be added automatically to suppress AFL's own UI).
- `timeout`: if not `None`, pass its value as the `-t` timeout parameter to AFL.
- `dictionary`: if not `None`, pass its value as the `-x` dictionary parameter to AFL.
- `master_name`: the name of the master fuzzer instance which will perform deterministic checks.

- `slave_name`: the name of a slave fuzzer instance which will proceed to random tweaks. For further details check: https://github.com/mirrorer/afl/blob/master/docs/parallel_fuzzing.txt

Example configuration snippet:

```
[sut.foo]
call=fuzzinator.call.SubprocessCall

[sut.foo.call]
command=./bin/foo {test}
cwd=/home/alice/foo
env={"BAR": "1"}

[fuzz.foo-with-afl]
sut=sut.foo
fuzzer=fuzzinator.fuzzer.AFLRunner
batch=inf
instances=1

[fuzz.foo-with-afl.fuzzer.init]
afl_fuzz=/home/alice/afl/afl-fuzz
sut_command=${sut.foo.call:command}
cwd=${sut.foo.call:cwd}
env=${sut.foo.call:env}
input=/home/alice/foo-inputs
output=${fuzzinator:work_dir}/afl-output/{uid}
```

5.2 class `ByteFlipDecorator`

class `fuzzinator.fuzzer.`**`ByteFlipDecorator`**(**args*, ***kwargs*)

Decorator to add extra random byte flips to fuzzer results.

Mandatory parameter of the decorator:

- `frequency`: the length of the test divided by this integer number gives the number of bytes flipped.

Optional parameters of the decorator:

- `min_byte`: minimum value for the flipped bytes (integer number, 32 by default, the smallest ASCII code of the printable characters).
- `max_byte`: maximum value for the flipped bytes (integer number, 126 by default, the largest ASCII code of the printable characters).

Example configuration snippet:

```
[sut.foo]
# see fuzzinator.call.*

[fuzz.foo-with-flips]
sut=sut.foo
fuzzer=fuzzinator.fuzzer.ListDirectory
fuzzer.decorate(0)=fuzzinator.fuzzer.ByteFlipDecorator
batch=inf

[fuzz.foo-with-flips.fuzzer.init]
outdir=/home/alice/foo-old-bugs/
```

```
[fuzz.foo-with-flips.fuzzer.decorate(0)]
frequency=100
min_byte=0
max_byte=255
```

5.3 class `FileWriterDecorator`

class `fuzzinator.fuzzer.`**`FileWriterDecorator`**(*filename*)

Decorator for fuzzers that create str or bytes-like output. The decorator writes the test input to a temporary file and replaces the output with the name of that file.

Mandatory parameter of the decorator:

- `filename`: path pattern for the temporary file, which may contain the substring `{uid}` as a placeholder for a unique string (replaced by the decorator).

Example configuration snippet:

```
[sut.foo]
# see fuzzinator.call.*

[fuzz.foo-with-random]
sut=sut.foo
fuzzer=fuzzinator.fuzzer.RandomContent
fuzzer.decorate(0)=fuzzionator.fuzzer.FileWriterDecorator

[fuzz.foo-with-random.fuzzer.decorate(0)]
filename=${fuzzinator:work_dir}/test-{uid}.txt
```

5.4 class `ListDirectory`

class `fuzzinator.fuzzer.`**`ListDirectory`**(*pattern*, *contents='True'*, ***kwargs*)

A simple test generator to iterate through existing files in a directory and return their contents one by one. Useful for re-testing previously discovered issues.

Since the fuzzer starts iterating from the beginning of the directory in every fuzz job, there is no gain in running multiple instances of this fuzzer in parallel. Because of the same reason, the fuzzer should be left running in the same fuzz job batch until all the files of the directory are processed.

Mandatory parameter of the fuzzer:

- `pattern`: shell-like pattern to the test files.

Optional parameter of the fuzzer:

- **`contents`: if it's true then the content of the files will be returned** instead of their path (boolean value, True by default).

Example configuration snippet:

```
[sut.foo]
# see fuzzinator.call.*

[fuzz.foo-with-oldbugs]
sut=sut.foo
```

```
fuzzer=fuzzinator.fuzzer.ListDirectory
instances=1
batch=inf

[fuzz.foo-with-oldbugs.fuzzer.init]
pattern=/home/alice/foo-old-bugs/**/*.js
```

5.5 function RandomContent

`fuzzinator.fuzzer.RandomContent(*, min_length='1', max_length='1', **kwargs)`

Example fuzzer to generate strings of random length from random ASCII uppercase letters and decimal digits.

Optional parameters of the fuzzer:

- `min_length`: minimum length of the string to generate (integer number, 1 by default)
- `max_length`: maximum length of the string to generate (integer number, 1 by default)

Example configuration snippet:

```
[sut.foo]
# see fuzzinator.call.*

[fuzz.foo-with-random]
sut=sut.foo
fuzzer=fuzzinator.fuzzer.RandomContent
batch=100

[fuzz.foo-with-random.fuzzer]
min_length=100
max_length=1000
```

5.6 class SubprocessRunner

`class fuzzinator.fuzzer.SubprocessRunner(outdir, command, cwd=None, env=None, timeout=None, contents='True', **kwargs)`

Wrapper around a fuzzer that is available as an executable and can generate its test cases as file(s) in a directory. First, the external executable is invoked as a subprocess, and once it has finished, the contents of the generated files are returned one by one.

Mandatory parameters of the fuzzer:

- `command`: string to pass to the child shell as a command to run (all occurrences of `{uid}` in the string are replaced by an identifier unique to this fuzz job).
- `outdir`: path to the directory containing the files generated by the external fuzzer (all occurrences of `{uid}` in the path are replaced by the same identifier as described at the `command` parameter).

Optional parameters of the fuzzer:

- `cwd`: if not `None`, change working directory before the command invocation.
- `env`: if not `None`, a dictionary of variable names-values to update the environment with.
- `timeout`: run subprocess with timeout.

- **`contents`: if it's true then the content of the files will be returned** instead of their path (boolean value, True by default).

Example configuration snippet:

```
[sut.foo]
# see fuzzinator.call.*

[fuzz.foo-with-bar]
sut=sut.foo
fuzzer=fuzzinator.fuzzer.SubprocessRunner
batch=50

[fuzz.foo-with-bar.fuzzer.init]
outdir=${fuzzinator:work_dir}/bar/{uid}
command=barfuzzer -n ${fuzz.foo-with-bar:batch} -o ${outdir}
```

5.7 class `TornadoDecorator`

class `fuzzinator.fuzzer.`**`TornadoDecorator`**(*port*, ***kwargs*)

Decorator for fuzzers to transport generated content through http. The decorator starts a Tornado server at the start of the fuzz job and returns a http url as test input. The SUT is expected to access the returned url and the decorated fuzzer is invoked on every GET access to that url. The response to the GET contains the generated test input prepended by a html meta tag to force continuous reloads in the SUT (or a `window.close()` javascript content to force stopping the SUT if the decorated fuzzer cannot generate more tests). Useful for transporting fuzz tests to browser SUTs.

Mandatory parameter of the fuzzer decorator:

- `port`: first port to start binding the started http server to (keeps incrementing until a free port is found).

Example configuration snippet:

```
[sut.foo]
# assuming that foo expects a http url as input, which it tries to access
# afterwards

[fuzz.foo-with-bar-over-http]
sut=sut.foo
#fuzzer=...
fuzzer.decorate(0)=fuzzinator.fuzzer.TornadoDecorator
batch=5

[fuzz.foo-with-bar-over-http.fuzzer.decorate(0)]
port=8000
```

CHAPTER 6

Test Case Reducers: package `fuzzinator.reduce`

6.1 function `Picire`

`fuzzinator.reduce.`**`Picire`**(*sut_call*, *sut_call_kwargs*, *listener*, *ident*, *issue*, *work_dir*, *parallel=False*, *combine_loops=False*, *split_method='zeller'*, *subset_first=True*, *subset_iterator='forward'*, *complement_iterator='forward'*, *jobs=4*, *max_utilization=100*, *encoding=None*, *atom='both'*, *granularity=2*, *cache_class='ContentCache'*, *cleanup=True*, ***kwargs*)

Test case reducer based on the Picire Parallel Delta Debugging Framework.

Optional parameters of the reducer:

- `parallel`, `combine_loops`, `split_method`, `subset_first`, `subset_iterator`, `complement_iterator`, `jobs`, `max_utilization`, `encoding`, `atom`, `granularity`, `cache_class`, `cleanup`

Refer to https://github.com/renatahodovan/picire for configuring Picire.

Note: This reducer is capable of detecting new issues found during the test reduction (if any).

Example configuration snippet:

```
[sut.foo]
#call=...
cost=1
reduce=fuzzinator.reduce.Picire
reduce_cost=4

[sut.foo.reduce]
parallel=True
jobs=4
subset_iterator=skip
```

6.2 function `Picireny`

`fuzzinator.reduce.`**`Picireny`**(*sut_call*, *sut_call_kwargs*, *listener*, *ident*, *issue*, *work_dir*, *hddmin=None*, *parallel=False*, *combine_loops=False*, *split_method='zeller'*, *subset_first=True*, *subset_iterator='forward'*, *complement_iterator='forward'*, *jobs=4*, *max_utilization=100*, *encoding=None*, *antlr=None*, *format=None*, *grammar=None*, *start=None*, *replacements=None*, *lang='python'*, *hdd_star=True*, *flatten_recursion=False*, *squeeze_tree=True*, *skip_unremovable=True*, *skip_whitespace=False*, *build_hidden_tokens=False*, *granularity=2*, *cache_class='ContentCache'*, *cleanup=True*, ***kwargs*)

Test case reducer based on the Picireny Hierarchical Delta Debugging Framework.

Mandatory parameters of the reducer:

- Either `format` or `grammar` and `start` must be defined.

Optional parameters of the reducer:

- `hddmin`, `parallel`, `combine_loops`, `split_method`, `subset_first`, `subset_iterator`, `complement_iterator`, `jobs`, `max_utilization`, `encoding`, `antlr`, `format`, `grammar`, `start`, `replacements`, `lang`, `hdd_star`, `flatten_recursion`, `squeeze_tree`, `skip_unremovable`, `skip_whitespace`, `build_hidden_tokens`, `granularity`, `cache_class`, `cleanup`

Refer to https://github.com/renatahodovan/picireny for configuring Picireny.

Note: This reducer is capable of detecting new issues found during the test reduction (if any).

Example configuration snippet:

```
[sut.foo]
#call=...
cost=1
reduce=fuzzinator.reduce.Picireny
reduce_cost=4

[sut.foo.reduce]
hddmin=full
grammar=/home/alice/grammars-v4/HTMLParser.g4 /home/alice/grammars-v4/
↪HTMLLexer.g4
start=htmlDocument
parallel=True
jobs=4
subset_iterator=skip
```

CHAPTER 7

SUT Updaters: package `fuzzinator.update`

7.1 function `SubprocessUpdate`

`fuzzinator.update.`**`SubprocessUpdate`**(*command*, *cwd=None*, *env=None*, *timeout=None*)

Subprocess invocation-based SUT update.

Mandatory parameter of the SUT update:

- `command`: string to pass to the child shell as a command to run.

Optional parameters of the SUT update:

- `cwd`: if not `None`, change working directory before the command invocation.
- `env`: if not `None`, a dictionary of variable names-values to update the environment with.
- `timeout`: run subprocess with timeout.

Example configuration snippet:

```
[sut.foo]
update=fuzzinator.update.SubprocessUpdate
#update_condition=... is needed to trigger the update

[sut.foo.update]
command=git pull && make
cwd=/home/alice/foo
env={"BAR": "1"}
```

7.2 function `TimestampUpdateCondition`

`fuzzinator.update.`**`TimestampUpdateCondition`**(*path*, *age*)

File timestamp-based SUT update condition.

Mandatory parameters of the SUT update condition:

- `path`: path to a file or directory to check for its last modification time.
- `age`: maximum allowed age of `path` given in [days:][hours:][minutes:]seconds format.

Result of the SUT update condition:

- Returns `True` if `path` does not exist or is older than `age`.

Example configuration snippet:

```
[sut.foo]
update_condition=fuzzinator.update.TimestampUpdateCondition
#update=... will be triggered if file timestamp is too old

[sut.foo.update_condition]
path=/home/alice/foo/bin/foo
age=7:00:00:00
```

CHAPTER 8

Release Notes

8.1 18.3.1

Summary of changes:

- Fixed the way package metadata is accessed to ensure wheel compatibility.

8.2 18.3

Summary of changes:

- New features in the framework:
  - Support for issue (re-)validation with a new job type (validate).
  - Support for user-defined event listeners.
- Numerous new building blocks in the framework:
  - `fuzzinator.EmailListener`: support for sending emails about events, e.g., new issues.
  - `fuzzinator.tracker.MonorailReport`: support for the Monorail issue tracking system.
  - `fuzzinator.call.LldbBacktraceDecorator`: support for backtrace info via LLDB.
  - `fuzzinator.fuzzer.ByteFlipDecorator`: support for adding extra random byte flips to fuzzer results.
  - `fuzzinator.fuzzer.FileWriterDecorator`: support for writing fuzzer results to files.
  - `fuzzinator.call.FileReaderDecorator`: support for extracting fuzzer results from files.
- Building blocks with extended or changed functionality:
  - `fuzzinator.call.SubprocessCall` and `.StdinSubprocessCall` can accept 0 exit code as an issue.

  - `fuzzinator.call.StreamRegexFilter` has been renamed to `.RegexFilter` to enable arbitrary filtering of issues, and can match multiple patterns.
  - `fuzzinator.call.StreamMonitoredSubprocessCall` regex patterns are multiline.
  - `fuzzinator.fuzzer.AFLRunner` supports AFL's master and slave concepts.
  - `fuzzinator.fuzzer.ListDirectory` works with a glob pattern instead of a simple directory name to collect test cases.
  - `fuzzinator.fuzzer.SubprocessRunner` and `.ListDirectory` can work both as file content generators and as file path generators.
  - `fuzzinator.update.TimestampUpdateCondition` supports time intervals longer than 24 hours.
  - All Popen-based subprocess-executing building blocks (`fuzzinator.call.StdinSubprocessCall`, `.StreamMonitoredSubprocessCall`, `.SubprocessCall`, `.SubprocessPropertyDecorator`, `.TestRunnerSubprocessCall`, `fuzzinator.fuzzer.SubprocessRunner`, and `fuzzinator.update.SubprocessUpdate`) have timeout support and avoid shell invocation.
  - `fuzzinator.reduce.Picire` has been updated to use *Picire* 18.1.
  - `fuzzinator.reduce.Picireny` has been updated to use *Picireny* 18.2.
  - All reporters (`fuzzinator.tracker.MonorailReport`, `.BugzillaReport`, `.GithubReport`) have been changed to use format string syntax ({key}) instead of template syntax ($key) in their report templates, and all handle missing keys gracefully.

- TUI improvements:
  - Support for simpler custom color schemes.
  - More convenient bug report editor.
  - Support for both text and binary copying of test cases to the clipboard.
  - Support for declaring bug duplicates manually.
  - New and improved dialogs (about dialog, closing of dialogs).
  - Improved event handling (responsivity, updated issues, invalid issues).
- General usability improvements:
  - More flexible configuration format enabling config sections to be split across multiple files, and keys to have no value.
  - Support for command line arguments specified in list files to help with config file fragments.
  - Useful command line argument aliases and new arguments (appearance, verbosity, Python interpreter limits, fuzz session length).
- Under-the-hood improvements:
  - Improved logging.
  - Added testing infrastructure: unit testing of SUT calls, fuzzers, and SUT updaters via tox; continuous testing via Travis and AppVeyor CI services.
  - Added documentation: out-of-sources tutorial and auto-generated API docs via Sphinx; online documentation hosting on Read-the-Docs.
  - Various bug fixes and refactorings (in core components, in building blocks, and in user interfaces).

8.3 16.10

First public release of the *Fuzzinator* Random Testing Framework.

Summary of main features:

- Core scheduler/controller of fuzzing-related jobs (update, fuzz, reduce).
- MongoDB-based issue repository.
- Extensible framework with predefined building blocks for invoking SUTs, detecting issues, and determining uniqueness; for generating test cases and transporting them to SUTs; for minimizing issue-triggering tests; and for keeping SUTs under development up-to-date.
- Configurability via INI files.
- CLI and Urwid-based TUI.

CHAPTER 9

Versioning and Releasing

9.1 Version Scheme

The project uses a date-based version scheme conforming to PEP440. The identifiers of official releases follow the "YY.MM" form (e.g., "16.10" for the version released on October, 2016), while development versions between two releases append an "r" suffix to the identifier of the last official release (e.g., "16.10r" for snapshots that contain changes on top of the "16.10" release).

(Alpha, beta, RC, and dev release version identifiers are not planned as of yet, as they would require the knowledge of the release date of the next offical release in advance - however, the project follows the "it will be released when it's ready, whenever that is" ideology.)

9.2 Commits in the Repository

For any official release, there should be exactly one commit in the repository that makes the project identify itself as the released version, and that commit should also be tagged with the version ID. Thus, the first commit after a release has to be a bump to an "r"-suffixed snapshot version.

9.3 Release Steps

The release of a new version happens along the following steps.

```
# name the new version and add release notes
echo "YY.MM" > fuzzinator/VERSION
nano RELNOTES.rst

# create a commit for the release, tag it, and push it to the public
# repository
git add fuzzinator/VERSION RELNOTES.rst
git commit -m "YY.MM release"
git tag YY.MM
git push origin master YY.MM

# upload the release to PyPI
python setup.py sdist upload -r pypi
```

Before landing anything in the repository after a release, the version should be bumped.

```
echo "YY.MMr" > fuzzinator/VERSION
git add fuzzinator/VERSION
git commit -m "Change to post-release version YY.MMr"
git push origin master
```

CHAPTER 10

Licensing

Python Module Index

Index