Regression Tests Documentation

Avast Software

Dec 07, 2022

Contents

This is a documentation for the regression tests framework. It describes how to write regression tests for RetDec and its tools, and provides an overview of the supported functions and methods.

The documentation is primarily focused on tests for the RetDec decompiler, i.e. for the `retdec-decompiler` script. You can, however, write tests for arbitrary RetDec-related tools, such as `fileinfo` or `unpacker`. This is described in section *Tests for Arbitrary Tools*. Nevertheless, it is highly recommended to read the whole documentation, even if you plan to write tests only for a single tool.

CHAPTER 1

Table of Contents

1.1 General Overview

This section contains a general overview of the directory structure for regression tests.

1.1.1 Basic Directory Structure

The *root directory* of all regression tests is `retdec-regression-tests`. The tests are structured into subdirectories, where each test is formed by a single subdirectory. Such a subdirectory has to contain:

- A `test.py` file, which is a *test module* containing test classes, settings, and methods that check whether the test passes or fails. Basically, it represents test specification.
- *Inputs* for the test. These are usually binary files to be decompiled.

A sample test structure looks like this:

```
regression-tests/
├── bugs/
│   └── invalid-function-name/
│       ├── input.exe
│       └── test.py
├── features/
│   └── main-detection/
│       ├── input.msvc.exe
│       ├── input.gcc.elf
│       └── test.py
└── integration/
        ├── ack/
        │   ├── ack.exe
        │   └── test.py
        └── ackermann/
            ├── ackermann.exe
            └── test.py
```

1.1.2 Outputs from Decompilations

After you run the regression tests, outputs from decompilations are placed into subdirectories named `outputs`. In this way, you can check them after the tests finish to see what exactly was generated. These subdirectories are created inside the corresponding test directory. For example, for `retdec-regression-tests/integration/ack`, the outputs are placed into `retdec-regression-tests/integration/ack/outputs`.

To differentiate between the outputs of different decompilations, the outputs from each decompilation are placed into a properly named directory. The name is based on the parameters that were passed to the decompiler when the test ran.

For example, a directory with outputs for the `factorial` test may contain the following subdirectories and files:

```
regression-tests/integration/factorial/outputs/
├── Test_2017 (factorial.x86.gcc.O0.exe)/
│   ├── factorial.x86.gcc.O0.c
│   ├── factorial.x86.gcc.O0.c-compiled
│   ├── factorial.x86.gcc.O0.bc
│   ├── factorial.x86.gcc.O0.ll
│   ├── factorial.x86.gcc.O0.c.fixed.c
│   ├── factorial.x86.gcc.O0.dsm
│   ├── factorial.x86.gcc.O0.config.json
│   └── factorial.x86.gcc.O0.c.log
├── ...
└── Test_2017 (factorial.x86.clang.O0.exe)/
    ├── factorial.x86.clang.O0.c
    ├── factorial.x86.clang.O0.c-compiled
    ├── factorial.x86.clang.O0.bc
    ├── factorial.x86.clang.O0.ll
    ├── factorial.x86.clang.O0.c.fixed.c
    ├── factorial.x86.clang.O0.dsm
    ├── factorial.x86.clang.O0.config.json
    └── factorial.x86.clang.O0.log
```

1.1.3 Naming Conventions

Directory names can contain characters that are valid on common file systems, **except a dot** (`.`). The reason for not allowing a dot is that it is internally used to separate subdirectories in module names. Python uses a dot to separate namespaces, anyway. Examples:

```
features                # OK
cool_backend_feature1   # OK
cool-backend-feature2   # OK

cool.backend.feature1   # WRONG: A dot '.' is not allowed by the regression tests␣
↪framework.
cool/backend/feature1   # WRONG: A slash '/' is not allowed in a directory name on␣
↪Linux.
cool|backend*feature1   # WRONG: Characters '|' and '*' are not allowed on Windows.
```

The naming of functions, classes, methods, and variables inside `test.py` files follows standard Python conventions, defined in PEP8. Example:

```
def my_function():
    my_var = [1, 2, 3]
```

(continues on next page)

(continued from previous page)

```
class MyClass(BaseClass):
    def do_something(self):
        ...

    def do_something_else(self):
        ...
```

1.1.4 Summary

To summarize, a *test* is a subdirectory that is arbitrarily nested inside the root directory and contains a `test.py` file and input files. After the tests are run, the output files are placed into a corresponding `outputs` directory. For every decompilation, the outputs are placed into a separate subdirectory.

Next, we will learn how to create a new test.

1.2 Creating a New Test

This section describes a way of creating new regression tests.

1.2.1 Basics

To create a new test, perform the following steps:

1. Create a new subdirectory inside the root directory. Example:

```
regression-tests
└── integration
    └── ack
```

The naming conventions are described in *General Overview*.

2. Create a `test.py` file inside that directory. Example:

```
regression-tests
└── integration
    └── ack
        └── test.py
```

The structure of this file is described later.

3. Add all the necessary input files to the created directory. Example:

```
regression-tests
└── integration
    └── ack
        ├── ack.exe
        └── test.py
```

1.2.2 General Structure of test.py Files

A `test.py` file has the following structure:

```
from regression_tests import *

# Test class 1
# Test class 2
# ...
```

The first line imports all the needed class names from the `regression_tests` package. You can import them explicitly if you want, but the `import *` is just fine in our case[1]. More specifically, it automatically imports the following classes, discussed later in this documentation:

- `Test`
- `TestSettings`

Also, other useful functions are automatically imported, which we will use later.

Every test class is of the following form:

```
class TestName(Test):
    # Settings 1
    # Settings 2
    # ...

    # Test method 1
    # Test method 2
    # ...
```

The name of a class can be any valid Python identifier, written by convention in `CamelCase`. Furthermore, it has to inherit from `Test`, which represents the base class of all tests. Example:

```
class AckTest(Test):
    # ...
```

Every test setting is of the following form:

```
settings_name = TestSettings(
    # Decompilation arguments.
    # ...
)
```

The arguments specify the input file(s), architecture(s), file format(s), and other settings to be used in the test cases.

Finally, there have to be some test methods to check that the decompiled code satisfies the needed properties. Every test method is of the following form:

```
def test_description_of_the_test(self):
    # Assertions
    # ...
```

A method has to start with the prefix `test_` and its name should contain a description of what exactly is this method testing. This is a usual convention among unit test frameworks. The reason is that when a test fails, its method name reveals the purpose of the test.

The assertions can be `assert` statements, assertions from the standard unittest module (`self.assertXYZ()`), or specific assertions provided by the classes of the regression tests framework. Example:

[1] Do not use the construct `import *` in real-world projects though because of namespace pollution and other reasons.

```
def test_ack_has_correct_signature(self):
    # The following statements assert that in the decompiled C code, there
    # is an ack() function, its return type is int, and that it has two
    # parameters of type int.
    self.assertTrue(self.out_c.funcs['ack'].return_type.is_int(32))
    self.assertEqual(self.out_c.funcs['ack'].param_count, 2)
    self.assertTrue(self.out_c.funcs['ack'].params[0].type.is_int(32))
    self.assertTrue(self.out_c.funcs['ack'].params[1].type.is_int(32))
```

The above example uses assertions from the standard unittest module[2].

See section *Writing Test Methods* for more details on how to write test methods and assertions.

1.2.3 Example of a test.py File

The following code is a complete example of a `test.py` file:

```
from regression_tests import *

class AckTest(Test):
    settings = TestSettings(
        input='ack.exe'
    )

    def test_ack_has_correct_signature(self):
        # The following statements assert that in the decompiled C code,
        # there is an ack() function, its return type is int, and that it
        # has two parameters of type int.
        self.assertTrue(self.out_c.funcs['ack'].return_type.is_int(32))
        self.assertEqual(self.out_c.funcs['ack'].param_count, 2)
        self.assertTrue(self.out_c.funcs['ack'].params[0].type.is_int(32))
        self.assertTrue(self.out_c.funcs['ack'].params[1].type.is_int(32))
```

Next, we will learn the details of specifying test settings.

1.3 Specifying Test Settings

This section gives a detailed description of how to specify settings for regression tests. It also describes the relation of test settings to test cases.

1.3.1 Basics

Test settings are specified as class attributes inside test classes, described in section *Creating a New Test*. The general form is

```
settings_name = TestSettings(
    arg1=value1,
    arg2=value2,
    # ...
)
```

[2] The assertions in the standard unittest module are historically named by using `CamelCase` instead of `snake_case`.

The name can by any valid Python identifier, written conventionally in `snake_case`. The arguments and values specified in the initializer of the used settings class define the parameters to be used for decompilations. For example, you may specify the input file or the used architecture. The selected arguments and values are then used to create arguments for the decompiler. For example, the following settings specify the input file and prescribe the use of the `x86` architecture:

```
settings = TestSettings(
    input='file.exe',
    arch='x86'
)
```

From the above settings, the following `retdec-decompiler` argument list is automatically created:

```
retdec-decompiler file.exe -a x86
```

For a complete list of possible arguments to the initializer, see the description of `DecompilerTestSettings`.

Every argument can be either a single value or a list of values. When you specify a single value, it will be used for all decompilations. However, if you specify multiple values, a separate decompilation is run for all of them. For example, consider the following test settings:

```
settings = TestSettings(
    input='file.exe',
    arch=['x86', 'arm']
)
```

For such settings, the following two decompilations are run:

```
retdec-decompiler file.exe -a x86
retdec-decompiler file.exe -a arm
```

That is, the regression tests framework runs a single decompilation for every combination of the values specified in the settings.

1.3.2 Test Cases and Their Creation

A *test case* is an instance of your test class with an associated decompilation. Recall that a test class is a class that inherits from `Test` (see section *Creating a New Test*). As described above, when you specify settings with some arguments having multiple values (e.g. several architectures), a separate decompilation is run for all of them. For every decompilation, a test case is created, and all its test methods are called.

For example, consider the following test:

```
class Sample(Test)
    settings = TestSettings(
        input='file.exe',
        arch=['x86', 'arm']
    )

    def test_something1(self):
        # ...

    def test_something2(self):
        # ...
```

For this test, the following two test cases are created:

```
Sample (file.exe -a x86)
Sample (file.exe -a arm)
```

The two test methods are then called on each of them:

```
Sample (file.exe -a x86)
    test_something1()
    test_something2()
Sample (file.exe -a arm)
    test_something1()
    test_something2()
```

1.3.3 Classes Having Multiple Settings

You may specify several settings in a test class. This is handy when you want to use different settings for some decompilations. For example, consider the following class:

```
class Sample(Test)
    settings1 = TestSettings(
        input='file1.exe',
        arch=['x86', 'arm']
    )
    settings2 = TestSettings(
        input='file2.elf',
        arch='thumb'
    )

    # Test methods...
```

We want to decompile `file1.exe` on `x86` and `arm`, and `file2.elf` on `thumb`. From this test class, the following three test cases are created:

```
Sample (file1.exe -a x86)
Sample (file1.exe -a arm)
Sample (file2.elf -a thumb)
```

1.3.4 Arbitrary Parameters for the Decompiler

If you look at the complete list of possible arguments (`DecompilerTestSettings`), you see that not all `retdec-decompiler` parameters may be specified as arguments to `TestSettings`. The reason is that `retdec-decompiler` provides too many parameters and their support in the form of arguments would be cumbersome. However, it is possible to specify arbitrary arguments that are directly passed to the `retdec-decompiler` via the `args` argument:

```
class Sample(Test):
    settings = TestSettings(
        input='file.exe'
        arch='x86',
        args='--select-decode-only --select-functions func1,func2'
    )
```

These settings result into the creation of the following decompilation:

```
retdec-decompiler file.exe -a x86 --select-decode-only --select-functions func1,func2
```

In a greater detail, the `args` argument is taken, split into sub-arguments by whitespace, and passed to the `retdec-decompiler`. The argument list internally looks like this:

```
['file.exe', '-a', 'x86', '--select-decode-only', '--select-functions', 'func1,func2']
```

Hint: When it is possible to specify a `retdec-decompiler` parameter in the form of a named argument (like architecture or endianness), always prefer it to specifying raw arguments by using the `args` argument. That is, do **not** write

```
class Sample(Test):
    settings = TestSettings(
        input='file.exe'
        args='-a x86'    # Always prefer using arch='x86'.
    )
```

The reason is that named arguments are less prone to changes in `retdec-decompiler`. Indeed, when such an argument changes in `retdec-decompiler`, all that has to be done is changing the internal mapping of named arguments to `retdec-decompiler` arguments. No test needs to be changed.

If you want to specify separate arguments for several decompilations for single settings, place them into a list when specifying the settings. For example, consider the following test class:

```
class Sample(Test):
    settings = TestSettings(
        input='file.elf'
        arch='x86',
        args=[
            '--select-decode-only --select-functions func1,func2',
            '--select-decode-only --select-functions func3'
        ]
    )
```

It results into these two decompilations:

```
retdec-decompiler file.elf -a x86 --select-decode-only --select-functions func1,func2
retdec-decompiler file.elf -a x86 --select-decode-only --select-functions func3
```

You can also specify multiple settings, as already described earlier in this section.

1.3.5 Specifying Input Files From Directory

When there is a lot of input files, the directory structure may become less readable (the `test.py` file is buried among input files). In a situation like this, you may put the input files into a directory (e.g. `input`) and use `files_in_dir()` to automatically generate a list of files in this directory:

```
settings = TestSettings(
    input=files_in_dir('inputs')
)
```

You can also specify which files should be included or excluded:

```
settings = TestSettings(
    input=files_in_dir('inputs', matching=r'.*\.exe', excluding=['problem-file.exe'])
)
```

1.4 Writing Test Methods

This section gives a description of how to write test methods. It also presents a brief overview of the available assertions.

1.4.1 Basics

Test methods are written in the same way as unit tests with the standard unittest module. Moreover, the base class for all tests, `Test`, inherits from unittest.TestCase. Therefore, you can use anything that is available in the `unittest` module, including assertion methods.

Every test method has to be begin with suffix `test`. Examples:

```
def test_something(self):
    # ...

def test_something_else(self):
    # ...
```

Its name should contain a description of what exactly is this method testing. This is the usual convention among unit test frameworks. The reason is that when a test fails, its method name reveals the purpose of the test.

Inside test methods, you can write your assertions about the decompiled code, or even about the decompilation itself. To write an assertion, you can either use the standard `assert` statement from Python, or you can use the assertions from the unittest module. For example, the following two assertions are functionally equivalent:

```
def test_ack_has_proper_number_of_parameters(self):
    # Using the assert statement:
    assert self.out_c.funcs['ack'].param_count == 2
    # Using a method from the unittest module:
    self.assertEqual(self.out_c.funcs['ack'].param_count, 2)
```

Hint: Even though both of these assertions are functionally equivalent, the assertions from the unittest module provide more useful output. For example, imagine that the decompiled function `ack()` has a different number of parameters than two. The first assertion fails with the following message:

```
FAIL: test_ack_has_proper_number_of_parameters (integration.ack.Test)
----------------------------------------------------------------------
Traceback (most recent call last):
File "integration/ack/test.py", line 34, in test_ack_has_proper_number_of_parameters
    assert self.out_c.funcs['ack'].param_count == 2
AssertionError
```

In contrast, the second assertion fails with this message:

```
FAIL: test_ack_has_proper_number_of_parameters (integration.ack.Test)
----------------------------------------------------------------------
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
File "integration/ack/test.py", line 36, in test_ack_has_proper_number_of_parameters
    self.assertEqual(self.out_c.funcs['ack'].param_count, 2)
AssertionError: 3 != 2
```

That is, you can see that the decompiled function has three parameters, but we were expecting two. Such a piece of information is not provided if you use the `assert` statement.

1.4.2 Automatically Performed Verifications

There are verifications that are performed automatically for every test case. Currently, there is only one such verification:

- The decompilation succeeded, i.e. it did not fail or timeout.

When the above verification fails (e.g. the decompilation timeouted), the test is ended with a failure without even running the test methods. Therefore, you do not have to manually test that the decompilation was successful.

Sometimes, however, you expect the decompilation to fail. For example, when the input file format is unsupported and you want to verify that a proper error message is emitted. In such cases, you can disable the automatic verification in the following way:

```
class Test(Test):
    settings = TestSettings(
        input='mips-elf64'
    )

    def setUp(self):
        # Prevent the base class from checking that the decompilation
        # succeeded (we expect it to fail).
        pass

    def test_decompilation_fails_with_correct_error(self):
        assert self.decompiler.return_code == 1
        assert self.decompiler.log.contains(r"Error: Unsupported target format 'ELF64
↪'.")
```

1.4.3 Accessing Decompilation and Outputs

The corresponding decompilation is available in the `self.decompiler` attribute, which is of type `Decompiler`. It provides access to the decompilation itself and to the produced files. Some of the available attributes are:

- `self.decompiler.args` - Arguments for the decompilation (`DecompilerArguments`).
- `self.decompiler.return_code` - Return code of the decompilation (`int`).
- `self.decompiler.timeouted` - Has the decompilation timeouted (`bool`)?
- `self.decompiler.log` - Full log from the decompilation (`Text`).
- `self.decompiler.fileinfo_output` - Output from fileinfo (`FileinfoOutput`).
- `self.decompiler.fileinfo_outputs` - A list of outputs from fileinfo (list of `FileinfoOutput`). This is handy when fileinfo ran multiple times (e.g. when the input file is packed).
- `self.decompiler.out_c` - Output C (`Module`).
- `self.decompiler.out_dsm` - Output DSM (`Text`).

- `self.decompiler.out_ll` - Output LLVM IR (`Text`).
- `self.decompiler.out_config` - Output configuration file (`Config`).

For a complete list of attributes `Decompiler` provides, see its documentation.

Hint: As a shortcut, the `Test` class provides the following attributes:

- `self.out_c` - Output C (alias to *self.decompiler.out_c*).
- `self.out_dsm` - Output DSM (alias to *self.decompiler.out_dsm*).
- `self.out_ll` - Output LLVM IR (alias to *self.decompiler.out_ll*).
- `self.out_config` - Output configuration file (alias to *self.decompiler.out_config*).

That is, you can write, e.g, `self.out_c` and `self.out_dsm` instead of `self.decompiler.out_c` and `self.decompiler.out_dsm`.

1.4.4 Accessing Test Settings

The settings used for the particular decompilation can be accessed through `self.settings`, which is of type `TestSettings`. This allows you to write assertions that depend on the input settings. Example:

```
if '--select-functions' in self.settings.args:
    # Specific assertions for selective decompilation.
    # ...
```

1.4.5 Verifying Text Outputs

Text outputs, like the decompilation log, output C, output LLVM IR, or output DSM, are instances of `Text`. Instances of this class behave like strings and provide all the methods the standard `str` class provides (documentation). Apart from them, however, it also contains a special method `contains()` that can be used to verify whether the output contains the given regular expression. Example:

```
assert self.out_c.contains(r'printf\("%d", \S+ / 4\);')
assert self.out_dsm.contains(r'; function: factorial')
assert self.out_ll.contains(r'dec_label_pc_400010')
assert self.decompiler.log.contains(r'# DONE')
```

1.4.6 Verifying C Outputs

Every C output is a text output. This means that all the methods of `Text` also work on C outputs. Nevertheless, C outputs, which are of class `Module`, provide a far richer interface for verifying the correctness of the decompiled C file. Indeed, as we see shortly, you can check whether the decompiled C contains the given functions, number of parameters, includes, comments, etc.

Available Entities

Class `Module` provides access to the following entities:

- the source code as a string (`self.out_c`, which is a string-like object),
- global variables (`Variable`),

- functions (`Function`),
- comments (`Comment`),
- includes (`Include`),
- string literals (`StringLiteral`),
- structures (`StructType`),
- unions (`UnionType`),
- enumerations (`EnumType`).

Supported types:

- void (`VoidType`),
- integral types (`IntegralType`), with subclasses `IntType`, `CharType`, and `BoolType`,
- floating-point types (`FloatingPointType`), with subclasses `FloatType` and `DoubleType`,
- pointers (`PointerType`),
- arrays (`ArrayType`),
- functions (`FunctionType`),
- structures (`StructType`),
- unions (`UnionType`),
- enumerations (`EnumType`).

Supported expressions:

- variables (`Variable`),
- literals (`Literal`), with subclasses `IntegralLiteral`, `FloatingPointLiteral`, `CharacterLiteral`, and `StringLiteral`,
- unary operators (`UnaryOpExpr`),
- binary operators (`BinaryOpExpr`),
- ternary operator (`TernaryOpExpr`),
- casts (`CastExpr`),
- function calls (`CallExpr`),
- initializer lists (`InitListExpr`).

Supported statements in function bodies:

- if (`IfStmt`),
- switch (`SwitchStmt`),
- for loop (`ForLoop`),
- while loop (`WhileLoop`),
- do-while loop (`DoWhileLoop`),
- break (`BreakStmt`),
- continue (`ContinueStmt`),
- return (`ReturnStmt`),

- goto (`GotoStmt`),
- empty statement (`EmptyStmt`),
- variable definition and assignment (`VarDefStmt`).

1.4.7 Verifying Configuration Files

Every output configuration file (in the JSON format) is a text file, so just like C files, you can use all the methods of `Text`. However, additional attributes and methods are provided to make the verification easier. See the documentation of `Config` for more details.

1.4.8 Compiling and Running C Outputs

To check that the generated C output is compilable, use `Test.assert_out_c_is_compilable()`:

```
self.assert_out_c_is_compilable()
```

To check that the output C produces the expected output when compiled and run, use `Test.assert_c_produces_output_when_run()`:

```
self.assert_c_produces_output_when_run('input text', 'expected output')
```

It compiles the output C file, runs it with the given input text, and verifies that the output matches.

Hint: By default, the compiled file is run with a 5 seconds timeout. If this value is insufficient, you can increase it by specifying an explicit timeout:

```
# Increase the timeout to 10 seconds.
self.assert_c_produces_output_when_run('input text', 'expected output', timeout=10)
```

1.4.9 Running Code Only On Selected Platforms

Some checks should run only on specific platforms (e.g. a test works only on Linux and Windows but not on macOS). The framework provides the following three functions to check on which operating system the test is running:

- `on_linux()`
- `on_macos()`
- `on_windows()`

Usage example:

```
if not on_macos():
    # This code will not be executed when running on macOS.
    # ...
```

There is no need to import anything (other than the usual *from regression_tests import ** at the top of a *test.py* file).

1.4.10 Examples

This sections contains examples of real-world assertions that you can use in your tests.

Output C

```
# Check that the output C is compilable.
self.assert_out_c_is_compilable()

# Check that the input and output C produce the same output
# when compiled and run with the given inputs:
self.assert_c_produces_output_when_run('0 0', 'ack(0, 0) = 1\n')
self.assert_c_produces_output_when_run('1 1', 'ack(1, 1) = 3\n')

# Check that there is at least one function.
assert self.out_c.has_funcs()

# Check that functions ack() and main() are present (possibly among other
# functions).
assert self.out_c.has_funcs('ack', 'main')

# If there are many functions, prefer the following way of testing of their
# presence. The reason is that when a function is missing, you immediately
# know which one is it.
assert self.out_c.has_func('ack')
assert self.out_c.has_func('main')

# You can also check that there is a function matching the given regular
# expression. This is useful if the exact function name can differ between
# platforms or compilers. For example, the name can be sometimes prefixed
# with an underscore.
assert self.out_c.has_func_matching(r'_?ack')

# Check that there are only two functions: ack() and main().
assert self.out_c.has_just_funcs('ack', 'main')

# Check that ack() calls itself recursively.
assert self.out_c.funcs['ack'].calls('ack')

# Check that there are no global variables.
assert self.out_c.has_no_global_vars()

# Check that there is the given string literal.
assert self.out_c.has_string_literal('Result is: %d');

# Check that there is a comment with the number of detected functions, and
# that the number of functions detected in the front-end matches the number
# of functions detected by the back-end.
assert self.out_c.has_comment_matching(r'.*Detected functions: (\d+) \(\1 in front-
↪end\)')

# Check that the following includes are present.
assert self.out_c.has_include_of_file('stdio.h')
assert self.out_c.has_include_of_file('stdint.h')

# Check that main has correct return type, parameter names and types.
assert self.out_c.funcs['main'].return_type.is_int()
assert self.out_c.funcs['main'].params[0].type.is_int()
self.assertEqual(self.out_c.funcs['main'].params[0].name, 'argc')
assert self.out_c.funcs['main'].params[1].type.is_pointer()
self.assertEqual(self.out_c.funcs['main'].params[1].name, 'argv')
```

(continues on next page)

(continued from previous page)

```
# Check that the ack() function has correct return type, parameter count,
# and types.
assert self.out_c.funcs['ack'].return_type.is_int(32)
self.assertEqual(self.out_c.funcs['ack'].param_count, 2)
assert self.out_c.funcs['ack'].params[0].type.is_int(32)
assert self.out_c.funcs['ack'].params[1].type.is_int(32)

# You can also use the following methods, which parse the given C type and
# check the correspondence.
assert self.out_c.funcs['ack'].params[1].type.is_same_as('int32_t')
assert self.out_c.funcs['ack'].params[1].type.is_same_as('int')
```

Output DSM

```
# Check that the output DSM contains the given comment.
assert self.out_dsm.contains(r'; function: ack')
```

Output LLVM IR

```
# Check that the output LLVM IR contains the given label.
assert self.out_ll.contains(r'dec_label_pc_400010')
```

Output Config

```
# Check that 10 functions are present in the config by using the JSON
# representation.
self.assertEqual(len(self.out_config.json.functions), 10)
```

Decompilation

```
# Check that the log from the decompilation contains
# the given front-end phase.
assert self.decompiler.log.contains(r'Running phase: libgcc idioms optimization')

# Check that the log from the decompilation contains
# the given comment.
assert self.decompiler.log.contains(r'# Done!')
```

In the next section, you will learn how to write tests for arbitrary tools, not just for the decompiler.

1.5 Tests for Arbitrary Tools

By default, when you specify test settings (see *Specifying Test Settings*), the tested tool is `decompiler`. Sometimes, however, it may be desirable to directly test other tools, such as `fileinfo` or `unpacker`, without the need to run a complete decompilation. This section describes how to write tests for tools other than the decompiler.

It is assumed that you have read all the previous sections. That is, to understand the present section, you need to know how to write tests and specify test settings for decompilations.

1.5.1 Specifying the Tool

To specify a tool that differs from the decompiler, use the `tool` parameter of `TestSettings`. For example, the next settings specify that `fileinfo` should be run with the given input file:

```
settings = TestSettings(
    tool='fileinfo',
    input='file.exe'
)
```

You can use any tool that is present in the RetDec's installation directory. For example, the following test settings prescribe to run all unit tests through the `retdec-tests-runner.py` script:

```
settings = TestSettings(
    tool='retdec-tests-runner.py'
)
```

Since this script does not take any inputs, the `input` parameter is omitted.

Note: If you do not explicitly specify a tool, `decompiler` is used. That is, the following two settings are equivalent:

```
settings1 = TestSettings(
    tool='decompiler',
    input='file.exe'
)

settings2 = TestSettings(
    input='file.exe'
)
```

1.5.2 Passing Parameters

As you have already learned, to specify the input file(s), use the `input` parameter of `TestSettings`. Remember that you can pass multiple input files at once:

```
settings = TestSettings(
    tool='fileinfo',
    input=['file1.exe', 'file2.exe', 'file3.exe']
)
```

When you do this, the tool is run separately for each of the input files.

Other parameters may be specified through the `args` parameter as a space-separated string. For example, to run `fileinfo` with two additional parameters `--json` and `--verbose`, use the following settings:

```
settings = TestSettings(
    tool='fileinfo',
    input='file.exe',
    args='--json --verbose'
)
```

Warning: You cannot specify parameters containing input or output files in this way (e.g. `-o file.out`). If you need to specify such parameters to test your tool, contact us.

Currently, the only supported tools with output files are `fileinfo` and `unpacker`. For them, the `-c` parameter (for `fileinfo`) or the `-o` parameter (for `unpacker`) are appended automatically by the regression-tests framework.

1.5.3 Obtaining Outputs and Writing Tests

As with tests for the decompiler, your tool is automatically run and the outputs are made available to you. To access them from your tests, use `self.$TOOL.$WHAT`. For example, for `fileinfo` tests, use `self.fileinfo.return_code` to get the return code or `self.fileinfo.output` to access the output.

For every tool, the following attributes are available:

- `self.$TOOL.return_code`: Return (exit) code from the tool (`int`).
- `self.$TOOL.timeouted`: Has the tool timeouted (`bool`)?
- `self.$TOOL.output`: Output from the tool (`Text`). It is a combined output from the standard and error outputs.

If your tool name includes characters out of `[a-zA-Z0-9_]`, all such characters are replaced with underscores. For example, for `retdec-tests-runner.py`, you would use `self.retdec_test_runner_sh.return_code` to get its return code.

Hint: This may be cumbersome to type, so for every tool, you can use the `self.tool.$WHAT` alias to access the particular output. That is, the following two types of access are equivalent:

```
assert self.retdec_test_runner_sh.return_code == 0
assert self.tool.return_code == 0
```

The `self.tool` alias is available for all tools.

1.5.4 Testing Success or Failure

Instead of checking the return code to verify success or failure, you can use the `succeeded` or `failed` attributes. For example, to verify that `fileinfo` ended successfully, use

```
assert self.fileinfo.succeeded
```

To check that `fileinfo` failed, use

```
assert self.fileinfo.failed
```

This makes tests more readable.

1.5.5 Tools with Extra Support

The regression-tests framework provides extra support for some tools. Such support may include additional test-settings parameters or attributes that can be used in tests. This section describes these tools. If you need additional support for your particular tool, contact us.

Fileinfo

- The output from `fileinfo` is parsed to provide easier access. See the description of `FileinfoOutput` for more details.
- Even though the tool's name is `fileinfo`, the tool is internally run via `retdec-fileinfo.py`. It is a shell script that wraps `retdec-fileinfo` and allows passing additional parameters. See its source code for more details.

IDA Plugin

- The tool's name is `idaplugin`. Internally, the tool is run via the `run-ida-decompilation.py` script.
- The input IDA database file can be specified via the `idb` parameter when creating test settings.
- The output C file can be accessed just like in decompilation tests: via `self.out_c`. All C checks are available (e.g. you can check that only the given function was decompiled).

Unpacker

- The `self.unpacker.succeeded` attribute checks not only that the return code is zero but also the existence of the unpacked file.
- It is possible to run `fileinfo` after unpacking. Just use the following parameter when constructing test settings: `run_fileinfo=True`. By default, `fileinfo` produces verbose output in the JSON format. For a complete list of supported `fileinfo`-related parameters, see the description of `UnpackerTestSettings`. You may then access the `fileinfo` run by accessing `self.fileinfo` (e.g. `self.fileinfo.output` gives you access to the parsed output).

1.5.6 Examples

Finally, we can take a look on some examples.

Bin2Pat

- The output YARA file generated by `bin2pat` is parsed to provide easier access to the YARA rules. See the description of `Yara` for more details.

Fileinfo

Regular `fileinfo` test.

```
class FileinfoTest(Test):
    settings = TestSettings(
        tool='fileinfo',
        input='sample.exe',
        args='--json' # JSON output is easier to parse than the default plain output.
    )

    def test_correctly_analyzes_input_file(self):
        assert self.fileinfo.succeeded

        self.assertEqual(self.fileinfo.output['architecture'], 'x86')
```

(continues on next page)

(continued from previous page)

```
        self.assertEqual(self.fileinfo.output['fileFormat'], 'PE')
        self.assertEqual(self.fileinfo.output['fileClass'], '32-bit')

        self.assertTrue(self.fileinfo.output['tools'][0]['name'].startswith('GCC'))
```

Verbose `fileinfo` with `loader_info` used.

```
class FileinfoTest(Test):
    settings = TestSettings(
        tool='fileinfo',
        args='-v',
        input='sample.exe'
    )

    def test_correctly_analyzes_input_file(self):
        assert self.fileinfo.succeeded

        self.assertEqual(self.fileinfo.output.loader_info.base_address, 0x400000)
        self.assertEqual(self.fileinfo.output.loader_info.loaded_segments_count, 2)

        self.assertEqual(self.fileinfo.output.loader_info.segments[0].name, '.text')
        self.assertEqual(self.fileinfo.output.loader_info.segments[0].address,␣
↪0x401000)
        self.assertEqual(self.fileinfo.output.loader_info.segments[0].size, 0x500)
        self.assertEqual(self.fileinfo.output.loader_info.segments[1].name, '.data')
```

IDA Plugin

```
class IDAPluginTest(Test):
    settings = TestSettings(
        tool='idaplugin',
        input='sample.exe',
        idb='sample.idb',
        args='--select 0x1000' # main
    )

    # Success is checked automatically (you do not have to check it manually).

    def test_correctly_decompiles_main(self):
        assert self.out_c.has_func('main')
```

Unpacker

```
class UnpackerTest(Test):
    settings = TestSettings(
        tool='unpacker',
        input='mpress.exe'
    )

    def test_correctly_unpacks_input_file(self):
        assert self.unpacker.succeeded
```

```
class UnpackerTest(Test):
    settings = TestSettings(
        tool='unpacker',
        input='upx.exe',
        run_fileinfo=True # Run fileinfo afterwards so we can check section names.
    )

    def test_unpacked_file_has_correct_section_names(self):
        assert self.unpacker.succeeded
        sections = self.fileinfo.output['sectionTable']['sections']
        self.assertEqual(len(sections), 4)
        self.assertEqual(sections[0]['name'], '.text')
        self.assertEqual(sections[1]['name'], '.rdata')
        self.assertEqual(sections[2]['name'], '.data')
        self.assertEqual(sections[3]['name'], '.reloc')
```

The next, last section concludes this documentation.

1.6 Support and Feedback

Congratulations! You have successfully reached the end of the API documentation (or you have just skipped to the end to see whether the butler did it). Anyhow, if you have any questions, remarks, or suggestions, feel free to contact us.

CHAPTER 2

Indices

- genindex
- modindex