Welcome to BinCFG's documentation!

BinCFG is a library to read, manipulate, and normalize binary Control Flow Graph (CFG) objects in python.

Installation Instructions

To install simply clone the repo:

git clone url/to/repo

Then make sure you are in your preferred python environment, and pip-install it:

pip install -e ./bincfg

NOTE: it is recommended to install with the *-e* flag as this installs the package in editable mode. A link to this *bincfg* directory is added to the environment's installed packages so that any changes made to the files in the *bincfg* directory are immediately reflected in any scripts using it. This way:

1. If you wish to alter behavior, or if there is a bug that needs a quick fix, you can do so without having to reinstall it with pip
2. If a new version of the code is pushed, you can simply *git pull* it and again not have to worry about reinstallation

Overview

This package allows users to load in output from binary analysis tools, and convert them into python CFG objects. For example, loading an output from the Rose binary analysis tool may look like:

```
from bincfg import CFG

cfg = CFG("/path/to/rose/cfg/file")
```

Now that you have a cfg object, you can print it to get some stats, or print its *.functions* attribute to print all the functions in this CFG:

```
print(cfg)
print()
print(cfg.functions)

# Output:
#
# CFG with no normalizer and 14 functions, 56 basic blocks, 65 edges, and 219 lines of assembly
#
# [CFGFunction innerfunc '_init' at 0x00001000 with 3 blocks, 8 asm lines , called by 1 basic blocks across 1
functions,
#  CFGFunction innerfunc with no name at 0x00001040 with 1 blocks, 2 asm lines , called by 1 basic blocks
across 1 functions,
#  CFGFunction externfunc 'printf@plt' at 0x00001050 with 1 blocks, 2 asm lines , called by 3 basic blocks
across 1 functions,
# ...
```

Often, you will wish to normalize the assembly lines in this cfg. This can be done using the `normalize_cfg_data()` function:

```
from bincfg.normalization import normalize_cfg_data

norm_cfg = normalize_cfg_data(cfg, 'innereye')

print(norm_cfg)
print()
print(cfg.functions[0].blocks)
print()
print(norm_cfg.functions[0].blocks)

# NOTE: this is using the builtin normalization method based on the 'innereye' method
#
# Output:
#
# CFG with normalizer: 'InnerEyeNormalizer' and 14 functions, 56 basic blocks, 65 edges, and 219 lines of
assembly
#
# [CFGBasicBlock in function "_init" at 0x00001000 unlabeled with 5 lines of assembly:
#     0x00001000: nop
#     0x00001004: sub    rsp, 0x08
#     0x00001008: mov    rax, qword ds:[rip + 0x0000000000002fd9<12249,absolute=0x0000000000003fe8>]
#     0x0000100f: test   rax, rax
#     0x00001012: je     0x0000000000001016<4118>,
#  CFGBasicBlock in function "_init" at 0x00001014 unlabeled with 1 lines of assembly:
#     0x00001014: call   rax,
#  CFGBasicBlock in function "_init" at 0x00001016 unlabeled with 2 lines of assembly:
#     0x00001016: add    rsp, 0x08
#     0x0000101a: ret]
#
# [CFGBasicBlock in function "_init" at 0x00001000 unlabeled with 5 lines of assembly:
#     0x00001000: ['nop']
#     0x00001004: ['sub_rsp_immval']
#     0x00001008: ['mov_rax_[_rip_+_immval_]']
#     0x0000100f: ['test_rax_rax']
#     0x00001012: ['je_immval'],
#  CFGBasicBlock in function "_init" at 0x00001014 unlabeled with 1 lines of assembly:
#     0x00001014: ['call_func'],
#  CFGBasicBlock in function "_init" at 0x00001016 unlabeled with 2 lines of assembly:
#     0x00001016: ['add_rsp_immval']
#     0x0000101a: ['ret']]
```

Basic blocks have now had their assembly lines normalized using the 'innereye' method.

This python representation of a CFG is relatively high-memory, inefficient, and not suitable for machine learning purposes. Instead, it is intended to be used only for loading in and parsing output from binary analysis tools, as well as providing an easier-to-traverse object if you wish to walk through the CFG.

For a more memory/space efficient, ML-ready representation of a CFG, you can convert it to a `MemCFG`:

```
from bincfg import MemCFG

mem_cfg = MemCFG(norm_cfg)

print(mem_cfg)

# Output:
#
# MemCFG with normalizer: 'InnerEyeNormalizer' and 14 functions, 56 blocks, 219 assembly lines, and 65 edges
```

NOTE: `MemCFG`'s require a normalization method. If you wish to use a `MemCFG` without doing any normalization, you can use the 'unnormalized' normalization provided by default by `BaseNormalizer`

Finally, you can get ML-ready data using `MemCFG` functions/attributes. For example, the `to_adjacency_matrix()` function will give you the adjacency matrix of that `MemCFG` in one of multiple common forms. See the `MemCFG` object for more info.

Contents:

Indices and tables