bincfg.labeling package

This subpackage provides utilities for labelling C/C++ basic blocks. It is still in development.

Submodules

bincfg.labeling.node_labels module

bincfg.labeling.node_labels.**DEFAULT_HEADER_PATH** = *'./NODE_LABEL_HEADER.h'*
Default path to save header file

bincfg.labeling.node_labels.**NODE_LABELS** = *['encryption', 'file_io', 'network_io', 'string_parser', 'error_handler']*
List of currently available node labels

bincfg.labeling.node_labels.**generate_c_header_file**(*path='./NODE_LABEL_HEADER.h'*)
Creates the C/C++ header file that will be imported in files that call our labelling functions.

Parameters: **path** (*str, optional*) – path to output file. Defaults to DEFAULT_HEADER_PATH.

bincfg.labeling.node_labels.**get_node_label**(*node_label*)
Gets the node label value from the given node label string

Parameters: **node_label** (*str*) – the string node label to get the value of. Case-insentitive.

Raises: **ValueError** – When an unknown node_label is passed

Returns: integer node label value (index of given string *node_label* in NODE_LABELS list)

Return type: int

bincfg.labeling.parse_cfg_labels module

Methods to help parse out CFG labels given a cfg that was produced on labeled source code.

labeling instructions:

1. Generate the .h header file for C/C++ code using the generate_c_header_file() function in bincfg.labeling.node_labels
2. Insert labels into the code at the start/stop locations of different labels.
    a. IE: for an encryption block, you can just insert the __NODE_LABEL_ENCRYPTION_START__ definition, and blocks will begin to be labeled 'encryption'
    b. Blocks start being labeled once you place a START instruction, and will continue being labeled until moving into another scope (EG: returning, entering a function call, exiting a subroutine), or until reaching the associated END instruction
    c. The instructions can only label at the basic-block level. Any basic block that contains one of these instructions will be given that label, regardless of where the instruction actually occurs in the block
    d. Blocks that have a START and END instruction within them will be given that instruction, but no more will be given after that block
    e. Labels will propagate to children. If a child has one labeled and one unlabeled parent, then it will be labeled
    f. Blocks can be given zero, one, or multiple labels
    g. It is recommended to place labeling instructions at the highest level possible to mitigate the chances that compiler optimizations will mess with labeling
    h. It is also recommended to place END instructions at the end of a function. Normally this isn't necessary, but if that function is inlined, then labels can propagate out of that function into places where it shouldn't be unless you have the END label at the end of that function
    i. You may also place a __CLEAR_NODE_LABELS__ instruction to end all node labels, except for those that are starting at that node. This is useful for functions that may be inlined if you wish for them to not have any other labels than those you purposefully give it. It's a matter of personal philosophy if, in this case, you think those blocks should inherit the labels of their inlining parents
3. Compile, analyze with rose, and load in with Bincfg
4. Now you have labeled basic blocks!

bincfg.labeling.parse_cfg_labels.**parse_node_labels**(*cfg*)
Parses node labels in the given cfg

Will check all blocks for start/end node labels and label basic blocks accordingly, then remove the labeling instructions from the block's assembly.

Assembly instructions for labeling should take the form of: "nop word ds:[LABEL_VAL]", where LABEL_VAL is the hex value for the label (label values will be in the generated .h header file, and should be in order of the NODE_LABELS list, one start and stop value for each).

Any nop instructions that do not fit that scheme, or that do not have a known LABEL_VAL, will be ignored.

CFGBasicBlock's will have their labels added to their .labels attribute: one int per label as a set

NOTE: this will not warn/error about mismatched END instructions, but it will raise an error for unknown nop labels

NOTE: This is implemented rather inefficiently. EG: all blocks with start instructions will be propagated, even if one of its parents already propagated that instruction

Pseudocode:

1. For every block that has a start instruction:
    a. Propagate that block as the ‘labeling_parent’ to all normal children recursively.
        - A ‘normal’ child is any child within the same function as the ‘labeling_parent’ block that can be reached through only normal edges
        - This way, we know which children blocks have a direct normal path up to the initial parent that has the start instruction
        - Only blocks with every parent being a descendent of the labeling_parent will be labeled, that way we only label blocks that are in the same scope
    b. Recursively propagate labels to all normal children
        - First check if that child should be labeled. IE: It is the labeling_parent block, or all parents of that block are themselves descendents of the labeling_parent block. We check that the _labeling_parent *is* the correct one so that we don’t have to do an extra pass to remove the _labeling_parent once labeling is complete
        - And, it should always have at least one label, otherwise the end instructions have removed all labels. It is done this way so that blocks that have both a start and an end instruction will still be given that label, but its children wont
2. For every block: remove any start/end nop labels. This way we (hopefully) get the exact same graph as without the labels

Parameters: **cfg** (*CFG*) – the cfg to parse node labels out of

Module contents