

362 lines 0 Removals

```

1 """
2 preprocessing: Implements the transformation from o
  bjdump output to ROMEO text representation.
3 """
4
5 import enum
6 import logging
7 import random
8 import re
9 import typing
10
11 import disassembler
12
13
14 class Granularity(enum.Enum):
15     INSTRUCTION = enum.auto()
16     FUNCTION = enum.auto()
17     TRANSLATION_UNIT = enum.auto()
18     OBJECT = enum.auto()
19
20
21 class LabelStrategy(enum.Enum):
22     BINARYCLASSIFICATION = enum.auto()
23     MULTICLASSCLASSIFICATION = enum.auto()
24
25
26 def validate_testcase_files(testcase) -> bool:
27     """This hook can be used in the future to add c
  riteria to testcases and stop the pipeline if they
  are not met."""
28     return True
29
30
31 def _build_symbol_translation_table(disassembly: li
  st[dict]) -> dict:
32     """Constructs a symbol translation table for th
  e given disassembly, assigning random names to non-
  dynamic functions."""
33     symbol_translation_table = {}
34     scrambled_symbol_count = 0
35     for name, symbol in disassembly["symbols"].item
  s():
36         # Local function names should be hidden!
37         if "F" in symbol["Flags"] and "D" not in sy
  mbol["Flags"]:
38             while True:
39                 proposed_translation = f"lc{random.
  randint(1,999):03d}"
40                 if proposed_translation not in symb
  ol_translation_table.values():
41                     symbol_translation_table[name]
  = proposed_translation
42                     break
43                 scrambled_symbol_count += 1
44                 if scrambled_symbol_count > 900:
45                     raise ValueError(
46                         f"Too many symbols to scramble
  for disassembly {disassembly['name']}"
47                     )
48             else:
49                 symbol_translation_table[name] = name
50
51     return symbol_translation_table
52

```

426 lines + 64 Additions

```

1 """
2 preprocessing: Implements the transformation from o
  bjdump output to ROMEO text representation.
3 """
4
5 import enum
6 import logging
7 import random
8 import re
9 import typing
10
11 import disassembler
12
13
14 class Granularity(enum.Enum):
15     INSTRUCTION = enum.auto()
16     FUNCTION = enum.auto()
17     TRANSLATION_UNIT = enum.auto()
18     OBJECT = enum.auto()
19
20
21 class LabelStrategy(enum.Enum):
22     BINARYCLASSIFICATION = enum.auto()
23     MULTICLASSCLASSIFICATION = enum.auto()
24
25
26 def validate_testcase_files(testcase) -> bool:
27     """This hook can be used in the future to add c
  riteria to testcases and stop the pipeline if they
  are not met."""
28     return True
29
30
31 def _build_symbol_translation_table(disassembly: li
  st[dict]) -> dict:
32     """Constructs a symbol translation table for th
  e given disassembly, assigning random names to non-
  dynamic functions."""
33     symbol_translation_table = {}
34     scrambled_symbol_count = 0
35     for name, symbol in disassembly["symbols"].item
  s():
36         # Local function names should be hidden!
37         if "F" in symbol["Flags"] and "D" not in sy
  mbol["Flags"]:
38             while True:
39                 proposed_translation = f"lc{random.
  randint(1,999):03d}"
40                 if proposed_translation not in symb
  ol_translation_table.values():
41                     symbol_translation_table[name]
  = proposed_translation
42                     break
43                 scrambled_symbol_count += 1
44                 if scrambled_symbol_count > 900:
45                     raise ValueError(
46                         f"Too many symbols to scramble
  for disassembly {disassembly['name']}"
47                     )
48             else:
49                 symbol_translation_table[name] = name
50
51     return symbol_translation_table
52

```

```

53
54 def _sanitize_symbol(symbol: str):
55     """Remove PLT artifacts etc. from symbols for e
asier matching."""
56     sanitized_symbol = str(symbol)
57
58     # 1 Remove multiple @ signs
59     while "@@" in sanitized_symbol:
60         sanitized_symbol = sanitized_symbol.replace
("@@", "@")
61
62     # 2 Remove things after @ sign
63     sanitized_symbol = sanitized_symbol.rsplit("@",
1)[0]
64
65     return sanitized_symbol
66
67
68 def _maybe_translate_symbol(
69     symbol: str, disassembly: list[dict], logger: l
ogging.Logger
70 ):
71     """Attemnt to use the symbol translation table t
o translate a symbol, otherwise return the original
symbol."""
72     symbol = _sanitize_symbol(symbol)
73     if symbol in disassembly["symbol_translation_ta
ble"].keys():
74         return disassembly["symbol_translation_tabl
e"][symbol]
75     else:
76         logger.warning(
77             f"Could not find symbol {symbol} in sym
bol translation table for disassembly {disassembly
['name']}"
78         )
79         return symbol
80
81
82 def _preprocess_disassembly(
83     disassembly: list[dict], logger: logging.Logger
84 ) -> list[dict]:
85     """Preprocesses a disassembly, replacing operan
d addresses with symbols and adding labels to funct
ions."""
86     disassembly["symbol_translation_table"] = _buil
d_symbol_translation_table(
87         disassembly
88     )
89
90     for _, section in disassembly["sections"].items
():
91         for function_name, function in dict(section
["functions"]).items():
92             instruction_text_representations = []
93             dependencies = []
94             for instruction in function["instructio
ns"]:
95                 instruction_text_representation: st
r = ""
96
97                 # 1 Operation
98                 instruction_text_representation +=
instruction["od_operation"]
99
100                # 2 Comment
101                if instruction["od_comment"] is not
None:

```

```

53
54 def _sanitize_symbol(symbol: str):
55     """Remove PLT artifacts etc. from symbols for e
asier matching."""
56     sanitized_symbol = str(symbol)
57
58     # 1 Remove multiple @ signs
59     while "@@" in sanitized_symbol:
60         sanitized_symbol = sanitized_symbol.replace
("@@", "@")
61
62     # 2 Remove things after @ sign
63     sanitized_symbol = sanitized_symbol.rsplit("@",
1)[0]
64
65     return sanitized_symbol
66
67
68 def _maybe_translate_symbol(
69     symbol: str, disassembly: list[dict], logger: l
ogging.Logger
70 ):
71     """Attemnt to use the symbol translation table t
o translate a symbol, otherwise return the original
symbol."""
72     symbol = _sanitize_symbol(symbol)
73     if symbol in disassembly["symbol_translation_ta
ble"].keys():
74         return disassembly["symbol_translation_tabl
e"][symbol]
75     else:
76         logger.warning(
77             f"Could not find symbol {symbol} in sym
bol translation table for disassembly {disassembly
['name']}"
78         )
79         return symbol
80
81
82 def _preprocess_disassembly(
83     disassembly: list[dict], logger: logging.Logger
84 ) -> list[dict]:
85     """Preprocesses a disassembly, replacing operan
d addresses with symbols and adding labels to funct
ions."""
86     disassembly["symbol_translation_table"] = _buil
d_symbol_translation_table(
87         disassembly
88     )
89
90     for _, section in disassembly["sections"].items
():
91         for function_name, function in dict(section
["functions"]).items():
92             instruction_text_representations = []
93             dependencies = []
94             for instruction in function["instructio
ns"]:
95                 instruction_text_representation: st
r = ""
96
97                 # 1 Operation
98                 instruction_text_representation +=
instruction["od_operation"]
99
100                # 2 Comment
101                if instruction["od_comment"] is not
None:

```

```

101         square_brackets = re.findall(
102             r"(?:QWORD PTR )?\[.+\]", instruction["od_operand"]
103         )
104         if len(square_brackets) == 1:
105             # Exactly one operand with
106             # an address, let's replace it with a comment
107             operand_to_replace = square_brackets[0]
108             instruction["od_operand"] = instruction["od_operand"].replace(
109                 operand_to_replace, instruction["od_comment"]
110             )
111             instruction["od_comment"] = None
112         else:
113             logger.warning(
114                 f"Could not uniquely match operand with comment for instruction: {str(instruction)}"
115             )
116             # 3 Operand
117             if instruction["od_operand"] is not None:
118                 instruction_text_representation += " " + instruction["od_operand"]
119             if (
120                 len(
121                     matches := re.findall(
122                         r"[0-9a-f]+ <.+>", instruction_text_representation
123                     )
124                 )
125                 > 0
126             ):
127                 if len(matches) != 1:
128                     logger.warning(
129                         f"Found multiple matches for instruction {instruction_text_representation} in function {function['name']} in section {section['name']} in disassembly {disassembly['name']}"
130                     )
131                 operand_in_question = matches[0]
132                 start_index = operand_in_question.index("<")
133                 end_index = operand_in_question.rfind(">")
134                 expression = operand_in_question[start_index + 1 : end_index]
135                 expression_offset_index = max(
136                     expression.rfind("+"),
137                     expression.rfind("-")
138                 )
139                 if (
140                     expression_offset_index
141                     != -1
142                     and re.match(
143                         r"[\+\-][0-9a-fx]", expression[expression_offset_index:]
144                     )

```

```

101         square_brackets = re.findall(
102             r"(?:QWORD PTR )?\[.+\]", instruction["od_operand"]
103         )
104         if len(square_brackets) == 1:
105             # Exactly one operand with
106             # an address, let's replace it with a comment
107             operand_to_replace = square_brackets[0]
108             instruction["od_operand"] = instruction["od_operand"].replace(
109                 operand_to_replace, instruction["od_comment"]
110             )
111             instruction["od_comment"] = None
112         else:
113             logger.warning(
114                 f"Could not uniquely match operand with comment for instruction: {str(instruction)}"
115             )
116             # 3 Operand
117             if instruction["od_operand"] is not None:
118                 instruction_text_representation += " " + instruction["od_operand"]
119             if (
120                 len(
121                     matches := re.findall(
122                         r"[0-9a-f]+ <.+>", instruction_text_representation
123                     )
124                 )
125                 > 0
126             ):
127                 if len(matches) != 1:
128                     logger.warning(
129                         f"Found multiple matches for instruction {instruction_text_representation} in function {function['name']} in section {section['name']} in disassembly {disassembly['name']}"
130                     )
131                 operand_in_question = matches[0]
132                 start_index = operand_in_question.index("<")
133                 end_index = operand_in_question.rfind(">")
134                 expression = operand_in_question[start_index + 1 : end_index]
135                 expression_offset_index = max(
136                     expression.rfind("+"),
137                     expression.rfind("-")
138                 )
139                 if (
140                     expression_offset_index
141                     != -1
142                     and re.match(
143                         r"[\+\-][0-9a-fx]", expression[expression_offset_index:]
144                     )

```



```

191         logger: logging.Logge
r,
192         ) -> list[dict]:
193     """Transform a testcase into a set of examples
    by disassembling and processing them."""
194     # First, disassemble all objects
195     object_files = [
196         file
197         for file in testcase["Files"]
198         if file["Path"].name.endswith(".o")
199         and not file["Path"].name.startswith("li
unked-")
200     ]
201     if not len(object_files) == 0:
202
203         object_file = object_files[0]
204         elf_path = object_file["Path"]
205
206         disassembled_code = disassembler.disassembl
e_elf([elf_path], logger)
207         disassembly = _preprocess_disassembly(disas
sembled_code, logger)
208
209         examples = []
210         if label_granularity == Granularity.FUNCTIO
N:
211             text_section_functions = disassembly["s
ections"][ ".text" ]["functions"]
212
213             # Decide which functions to use as exam
ples
214             primary_good_functions = []
215             primary_bad_functions = []
216             secondary_good_functions = []
217
218             if str(disassembly['name']).endswith("_
good.o"):
219                 primary_good_functions.append(disas
sembly['name'])
220             else:
221                 primary_bad_functions.append(disass
sembly['name'])
222
223             # Emit labeled functions
224             emitted_functions = secondary_good_func
tions + primary_bad_functions
225             if emit_primary_good_function:
226                 emitted_functions += primary_good_f
unctions
227
228             for function_name in emitted_functions:
229                 function = text_section_functions
["_start"]
230
231                 text_representation = ""
232                 for text_section in text_section_fu
nctions:
233                     text_representation += text_sec
tion_functions[text_section]["TextRepresentation"]
234                     example = {
235                         "Example": text_representation,
236                         "Testcase": testcase,
237                         "GoodOrBad": "Good"
238                     }
239                     if function_name not in primary
_bad_functions
240                     else "Bad",
241                     }
242                     examples += [example]

```

```

187 def extract_examples(
188     testcase: dict,
189     label_granularity: typing.Optional[Granularit
190 y],
191     context_granularity: typing.Optional[Granularit
192 y],
193     emit_primary_good_function: bool,
194     logger: logging.Logger,
195 ) -> list[dict]:
196     """Transform a testcase into a set of examples
197     by disassembling and processing them."""
198     # First, disassemble all objects
199     object_files = [
200         file
201         for file in testcase["Files"]
202         if file["Path"].name.endswith(".o")
203         and not file["Path"].name.startswith("linke
204 d-")
205     ]
206     disassembly = _preprocess_disassembly(
207         disassembler.disassemble_elf(
208             [object_file["Path"] for object_file in
209 object_files], logger
210         ),
211         logger,
212     )
213     examples = []
214     if label_granularity == Granularity.FUNCTION:
215         text_section_functions = disassembly["secti
216 ons"][ ".text" ]["functions"]
217
218         # Decide which functions to use as examples
219         primary_good_functions = []
220         primary_bad_functions = []
221         secondary_good_functions = []
222
223         for function_name in text_section_function
224 s:
225             if re.match(r"^(CWE.*(_|:))bad(\\(\\))?
226 $", function_name) is not None:
227                 primary_bad_functions += [function_
228 name]
229             elif re.match(r"^(CWE.*(_|:))good(\\
230 (\\))?$", function_name) is not None:
231                 primary_good_functions += [function
232 _name]
233             elif (
234                 re.match(
235                     r"^(CWE.*(_|:))good(\\d+|G2B\\d
236 *|B2G\\d*)(\\(\\))?$", function_name
237                 )
238                 is not None
239             ):

```

```

241
242         else:
243             raise ValueError("Granularity not suppo
244 rted")
245
246         return examples
247     else:
248         print("No Object Files")
249         exit(1)
250
251 def extract_examples(
252     testcase: dict,
253     label_granularity: typing.Optional[Granularit
254 y],
255     context_granularity: typing.Optional[Granularit
256 y],
257     emit_primary_good_function: bool,
258     logger: logging.Logger,
259 ) -> list[dict]:
260     """Transform a testcase into a set of examples
261     by disassembling and processing them."""
262     # First, disassemble all objects
263     object_files = [
264         file
265         for file in testcase["Files"]
266         if file["Path"].name.endswith(".o")
267         and not file["Path"].name.startswith("linke
268 d-")
269     ]
270     disassembly = _preprocess_disassembly(
271         disassembler.disassemble_elf(
272             [object_file["Path"] for object_file in
273 object_files], logger
274         ),
275         logger,
276     )
277     examples = []
278     if label_granularity == Granularity.FUNCTION:
279         text_section_functions = disassembly["secti
280 ons"][ ".text" ]["functions"]
281
282         # Decide which functions to use as examples
283         primary_good_functions = []
284         primary_bad_functions = []
285         secondary_good_functions = []
286
287         for function_name in text_section_function
288 s:
289             if re.match(r"^(CWE.*(_|:))bad(\\(\\))?
290 $", function_name) is not None:
291                 primary_bad_functions += [function_
292 name]
293             elif re.match(r"^(CWE.*(_|:))good(\\
294 (\\))?$", function_name) is not None:
295                 primary_good_functions += [function
296 _name]
297             elif (
298                 re.match(
299                     r"^(CWE.*(_|:))good(\\d+|G2B\\d
300 *|B2G\\d*)(\\(\\))?$", function_name
301                 )
302                 is not None
303             ):

```

```

230         secondary_good_functions += [function_name]
231
232         # See if all functions are there
233
234         # Ignore missing good functions for "Bad-Only Test Cases" (see Appendix D in Juliet 1.2 doc)
235         if (
236             testcase["Weakness"]["WeaknessID"] != 5
237             or testcase["FunctionalVariant"]
238             not in [
239                 "email",
240                 "file_transfer_connect_socket",
241                 "file_transfer_listen_socket",
242                 "screen_capture",
243             ]
244             ) and (
245             testcase["Weakness"]["WeaknessID"] != 5
246             or testcase["FunctionalVariant"]
247             not in ["network_connection", "network_listen"]
248             ):
249             if not (len(primary_good_functions) > 0) and (
250                 len(secondary_good_functions) > 0
251             ):
252                 logger.warning(
253                     f"Number of primary or secondary good functions is zero for testcase {str(testcase)}!"
254                 )
255
256             if not (len(primary_bad_functions) > 0):
257                 logger.warning(
258                     f"Number of primary bad functions is zero for testcase {str(testcase)}!"
259                 )
260
261             # Emit labeled functions
262             emitted_functions = secondary_good_functions + primary_bad_functions
263             if emit_primary_good_function:
264                 emitted_functions += primary_good_functions
265
266             for function_name in emitted_functions:
267                 function = text_section_functions[function_name]
268
269                 example = {
270                     "Example": function["TextRepresentation"],
271                     "Testcase": testcase,
272                     "GoodOrBad": "Good"
273                 }
274                 if function_name not in primary_bad_functions:
275                     else "Bad",
276                 }
277                 if context_granularity == Granularity.FUNCTION:
278                     pass
279                 elif context_granularity == Granularity.OBJECT:

```

```

294         secondary_good_functions += [function_name]
295
296         # See if all functions are there
297
298         # Ignore missing good functions for "Bad-Only Test Cases" (see Appendix D in Juliet 1.2 doc)
299         if (
300             testcase["Weakness"]["WeaknessID"] != 5
301             or testcase["FunctionalVariant"]
302             not in [
303                 "email",
304                 "file_transfer_connect_socket",
305                 "file_transfer_listen_socket",
306                 "screen_capture",
307             ]
308             ) and (
309             testcase["Weakness"]["WeaknessID"] != 5
310             or testcase["FunctionalVariant"]
311             not in ["network_connection", "network_listen"]
312             ):
313             if not (len(primary_good_functions) > 0) and (
314                 len(secondary_good_functions) > 0
315             ):
316                 logger.warning(
317                     f"Number of primary or secondary good functions is zero for testcase {str(testcase)}!"
318                 )
319
320             if not (len(primary_bad_functions) > 0):
321                 logger.warning(
322                     f"Number of primary bad functions is zero for testcase {str(testcase)}!"
323                 )
324
325             # Emit labeled functions
326             emitted_functions = secondary_good_functions + primary_bad_functions
327             if emit_primary_good_function:
328                 emitted_functions += primary_good_functions
329
330             for function_name in emitted_functions:
331                 function = text_section_functions[function_name]
332
333                 example = {
334                     "Example": function["TextRepresentation"],
335                     "Testcase": testcase,
336                     "GoodOrBad": "Good"
337                 }
338                 if function_name not in primary_bad_functions:
339                     else "Bad",
340                 }
341                 if context_granularity == Granularity.FUNCTION:
342                     pass
343                 elif context_granularity == Granularity.OBJECT:

```

```

279         # Ignore functions that are not hel
pful or leak labels
280         ignored_functions = [
281             "_start",
282             "__libc_csu_init",
283             "__libc_csu_fini",
284             "_dl_relocate_static_pie",
285             "deregister_tm_clones",
286             "register_tm_clones",
287             "__do_global_dtors_aux",
288             "frame_dummy",
289         ] + [function_name]
290
291         if function_name in primary_bad_fun
ctions:
292             ignored_functions += (
293                 primary_good_functions + se
condary_good_functions
294             )
295         else:
296             if not (
297                 function_name in primary_go
od_functions
298                 or function_name in seconda
ry_good_functions
299             ):
300                 logger.warning(
301                     f"Function {function_na
me} not found in primary or secondary good function
s or bad functions for testcase {str(testcase)}!"
302                 )
303                 logger.warning(
304                     f"Candidate functions:
{primary_good_functions}+{secondary_good_function
s}+{primary_bad_functions}"
305                 )
306                 ignored_functions += primary_ba
d_functions
307
308             # Build list of candidate functions
309             candidate_functions = {function_nam
e}
310             while True:
311                 new_candidate_functions = set(c
andidate_functions)
312                 for candidate_function in candi
date_functions:
313                     for dependency in text_sect
ion_functions[candidate_function][
314                         "Dependencies"
315                     ]:
316                         if dependency in text_s
ection_functions.keys():
317                             new_candidate_func
tions.add(dependency)
318                             if len(new_candidate_functions)
== len(candidate_functions):
319                                 break
320                             else:
321                                 candidate_functions = new_c
andidate_functions
322
323                             context_functions = [
324                                 fnn for fnn in candidate_func
tions if fnn not in ignored_functions
325                             ]

```

```

343         # Ignore functions that are not hel
pful or leak labels
344         ignored_functions = [
345             "_start",
346             "__libc_csu_init",
347             "__libc_csu_fini",
348             "_dl_relocate_static_pie",
349             "deregister_tm_clones",
350             "register_tm_clones",
351             "__do_global_dtors_aux",
352             "frame_dummy",
353         ] + [function_name]
354
355         if function_name in primary_bad_fun
ctions:
356             ignored_functions += (
357                 primary_good_functions + se
condary_good_functions
358             )
359         else:
360             if not (
361                 function_name in primary_go
od_functions
362                 or function_name in seconda
ry_good_functions
363             ):
364                 logger.warning(
365                     f"Function {function_na
me} not found in primary or secondary good function
s or bad functions for testcase {str(testcase)}!"
366                 )
367                 logger.warning(
368                     f"Candidate functions:
{primary_good_functions}+{secondary_good_function
s}+{primary_bad_functions}"
369                 )
370                 ignored_functions += primary_ba
d_functions
371
372             # Build list of candidate functions
373             candidate_functions = {function_nam
e}
374             while True:
375                 new_candidate_functions = set(c
andidate_functions)
376                 for candidate_function in candi
date_functions:
377                     for dependency in text_sect
ion_functions[candidate_function][
378                         "Dependencies"
379                     ]:
380                         if dependency in text_s
ection_functions.keys():
381                             new_candidate_func
tions.add(dependency)
382                             if len(new_candidate_functions)
== len(candidate_functions):
383                                 break
384                             else:
385                                 candidate_functions = new_c
andidate_functions
386
387                             context_functions = [
388                                 fnn for fnn in candidate_func
tions if fnn not in ignored_functions
389                             ]

```



```

326         context = "\n".join(
327             [
328                 text_section_functions[fnn]
329                 ["TextRepresentation"]
330                 for fnn in context_function
331             ]
332         )
333         example["Example"] += "\n" + context
334
335         elif context_granularity is None:
336             pass
337         else:
338             raise ValueError("Granularity not supported for context")
339
340         examples += [example]
341
342     else:
343         raise ValueError("Granularity not supported")
344
345     return examples
346
347 def label_example(example, label_strategy: LabelStrategy):
348     """Compute a label for a given example depending on the label strategy."""
349     if label_strategy == LabelStrategy.BINARYCLASSIFICATION:
350         return example["GoodOrBad"]
351     elif label_strategy == LabelStrategy.MULTICLASSCLASSIFICATION:
352         if example["GoodOrBad"] == "Good":
353             return "NoWeakness"
354         else:
355             return f"CWE{example['Testcase']['Weakness']['WeaknessID']}"
356     else:
357         raise ValueError("Label strategy not supported")
358
359 __all__ = ["extract_labeled_examples", "validate_testcase_files", "label_example"]
360
361
362

```

```

390         context = "\n".join(
391             [
392                 text_section_functions[fnn]
393                 ["TextRepresentation"]
394                 for fnn in context_function
395             ]
396         )
397         example["Example"] += "\n" + context
398
399         elif context_granularity is None:
400             pass
401         else:
402             raise ValueError("Granularity not supported for context")
403
404         examples += [example]
405
406     else:
407         raise ValueError("Granularity not supported")
408
409     return examples
410
411 def label_example(example, label_strategy: LabelStrategy):
412     """Compute a label for a given example depending on the label strategy."""
413     if label_strategy == LabelStrategy.BINARYCLASSIFICATION:
414         return example["GoodOrBad"]
415     elif label_strategy == LabelStrategy.MULTICLASSCLASSIFICATION:
416         if example["GoodOrBad"] == "Good":
417             return "NoWeakness"
418         else:
419             return f"CWE{example['Testcase']['Weakness']['WeaknessID']}"
420     else:
421         raise ValueError("Label strategy not supported")
422
423 __all__ = ["extract_labeled_examples", "validate_testcase_files", "label_example"]
424
425
426

```