362 lines 0 Removals

```
"""
preprocessing: Implements the transformation from objdump output to ROMEO text representation.
"""

import enum
import logging
import random
import re
import typing

import disassembler


class Granularity(enum.Enum):
    INSTRUCTION = enum.auto()
    FUNCTION = enum.auto()
    TRANSLATION_UNIT = enum.auto()
    OBJECT = enum.auto()


class LabelStrategy(enum.Enum):
    BINARYCLASSIFICATION = enum.auto()
    MULTICLASSCLASSIFICATION = enum.auto()


def validate_testcase_files(testcase) -> bool:
    """This hook can be used in the future to add criteria to testcases and stop the pipeline if they are not met."""
    return True


def _build_symbol_translation_table(disassembly: list[dict]) -> dict:
    """Constructs a symbol translation table for the given disassembly, assigning random names to non-dynamic functions."""
    symbol_translation_table = {}
    scrambled_symbol_count = 0
    for name, symbol in disassembly["symbols"].items():
        # Local function names should be hidden!
        if "F" in symbol["Flags"] and "D" not in symbol["Flags"]:
            while True:
                proposed_translation = f"lc{random.randint(1,999):03d}"
                if proposed_translation not in symbol_translation_table.values():
                    symbol_translation_table[name] = proposed_translation
                    break
            scrambled_symbol_count += 1
            if scrambled_symbol_count > 900:
                raise ValueError(
                    f"Too many symbols to scramble for disassembly {disassembly['name']}"
                )
        else:
            symbol_translation_table[name] = name

    return symbol_translation_table

```

426 lines + 64 Additions

```
"""
preprocessing: Implements the transformation from objdump output to ROMEO text representation.
"""

import enum
import logging
import random
import re
import typing

import disassembler


class Granularity(enum.Enum):
    INSTRUCTION = enum.auto()
    FUNCTION = enum.auto()
    TRANSLATION_UNIT = enum.auto()
    OBJECT = enum.auto()


class LabelStrategy(enum.Enum):
    BINARYCLASSIFICATION = enum.auto()
    MULTICLASSCLASSIFICATION = enum.auto()


def validate_testcase_files(testcase) -> bool:
    """This hook can be used in the future to add criteria to testcases and stop the pipeline if they are not met."""
    return True


def _build_symbol_translation_table(disassembly: list[dict]) -> dict:
    """Constructs a symbol translation table for the given disassembly, assigning random names to non-dynamic functions."""
    symbol_translation_table = {}
    scrambled_symbol_count = 0
    for name, symbol in disassembly["symbols"].items():
        # Local function names should be hidden!
        if "F" in symbol["Flags"] and "D" not in symbol["Flags"]:
            while True:
                proposed_translation = f"lc{random.randint(1,999):03d}"
                if proposed_translation not in symbol_translation_table.values():
                    symbol_translation_table[name] = proposed_translation
                    break
            scrambled_symbol_count += 1
            if scrambled_symbol_count > 900:
                raise ValueError(
                    f"Too many symbols to scramble for disassembly {disassembly['name']}"
                )
        else:
            symbol_translation_table[name] = name

    return symbol_translation_table

```

```

def _sanitize_symbol(symbol: str):
    """Remove PLT artifacts etc. from symbols for easier matching."""
    sanitized_symbol = str(symbol)

    # 1 Remove multiple @ signs
    while "@@" in sanitized_symbol:
        sanitized_symbol = sanitized_symbol.replace("@@", "@")

    # 2 Remove things after @ sign
    sanitized_symbol = sanitized_symbol.rsplit("@", 1)[0]

    return sanitized_symbol


def _maybe_translate_symbol(
    symbol: str, disassembly: list[dict], logger: logging.Logger
):
    """Attemt to use the symbol translation table to translate a symbol, otherwise return the original symbol."""
    symbol = _sanitize_symbol(symbol)
    if symbol in disassembly["symbol_translation_table"].keys():
        return disassembly["symbol_translation_table"][symbol]
    else:
        logger.warning(
            f"Could not find symbol {symbol} in symbol translation table for disassembly {disassembly['name']}"
        )
        return symbol


def _preprocess_disassembly(
    disassembly: list[dict], logger: logging.Logger
) -> list[dict]:
    """Preprocesses a disassembly, replacing operand addresses with symbols and adding labels to functions."""
    disassembly["symbol_translation_table"] = _build_symbol_translation_table(
        disassembly
    )

    for _, section in disassembly["sections"].items():
        for function_name, function in dict(section["functions"]).items():
            instruction_text_representations = []
            dependencies = []
            for instruction in function["instructions"]:
                instruction_text_representation: str = ""
                # 1 Operation
                instruction_text_representation += instruction["od_operation"]

                # 2 Comment
                if instruction["od_comment"] is not None:
```


```

def _sanitize_symbol(symbol: str):
    """Remove PLT artifacts etc. from symbols for easier matching."""
    sanitized_symbol = str(symbol)

    # 1 Remove multiple @ signs
    while "@@" in sanitized_symbol:
        sanitized_symbol = sanitized_symbol.replace("@@", "@")

    # 2 Remove things after @ sign
    sanitized_symbol = sanitized_symbol.rsplit("@", 1)[0]

    return sanitized_symbol


def _maybe_translate_symbol(
    symbol: str, disassembly: list[dict], logger: logging.Logger
):
    """Attemt to use the symbol translation table to translate a symbol, otherwise return the original symbol."""
    symbol = _sanitize_symbol(symbol)
    if symbol in disassembly["symbol_translation_table"].keys():
        return disassembly["symbol_translation_table"][symbol]
    else:
        logger.warning(
            f"Could not find symbol {symbol} in symbol translation table for disassembly {disassembly['name']}"
        )
        return symbol


def _preprocess_disassembly(
    disassembly: list[dict], logger: logging.Logger
) -> list[dict]:
    """Preprocesses a disassembly, replacing operand addresses with symbols and adding labels to functions."""
    disassembly["symbol_translation_table"] = _build_symbol_translation_table(
        disassembly
    )

    for _, section in disassembly["sections"].items():
        for function_name, function in dict(section["functions"]).items():
            instruction_text_representations = []
            dependencies = []
            for instruction in function["instructions"]:
                instruction_text_representation: str = ""
                # 1 Operation
                instruction_text_representation += instruction["od_operation"]

                # 2 Comment
                if instruction["od_comment"] is not None:
```

```
                    square_brackets = re.findall(
                        r"(?:QWORD PTR )?\[.+\]", instruction["od_operand"]
                    )
                    if len(square_brackets) == 1:
                        # Exactly one operand with an address, let's replace it with a comment
                        operand_to_replace = square_brackets[0]
                        instruction["od_operand"] = instruction["od_operand"].replace(
                            operand_to_replace, instruction["od_comment"]
                        )
                        instruction["od_comment"] = None
                    else:
                        logger.warning(
                            f"Could not uniquely match operand with comment for instruction: {str(instruction)}"
                        )

                # 3 Operand
                if instruction["od_operand"] is not None:
                    instruction_text_representation += " " + instruction["od_operand"]

                    if (
                        len(
                            matches := re.findall(
                                r"[0-9a-f]+ <.+>", instruction_text_representation
                            )
                        )
                        > 0
                    ):
                        if len(matches) != 1:
                            logger.warning(
                                f"Found multiple matches for instruction {instruction_text_representation} in function {function['name']} in section {section['name']} in disassembly {disassembly['name']}"
                            )
                        operand_in_question = matches[0]
                        start_index = operand_in_question.index("<")
                        end_index = operand_in_question.rfind(">")
                        expression = operand_in_question[start_index + 1 : end_index]

                        expression_offset_index = max(
                            expression.rfind("+"), expression.rfind("-")
                        )
                        if (
                            expression_offset_index != -1
                            and re.match(
                                r"[\+\-][0-9a-fx]", expression[expression_offset_index:]
                            )
```

```
                            is not None
                        ):
                            offset = expression[expression_offset_index:]
                        else:
                            expression_offset_index = -1
                            offset = ""

                        symbol_end_index = (
                            expression_offset_index
                            if expression_offset_index != -1
                            else len(expression)
                        )

                        symbol = expression[:symbol_end_index]
                        dependencies += [_sanitize_symbol(symbol)]

                        symbol = _maybe_translate_symbol(symbol, disassembly, logger)

                        instruction_text_representation = (
                            instruction_text_representation.replace(
                                operand_in_question, f"{symbol}{offset}"
                            )
                        )

                instruction_text_representations += [
                    instruction_text_representation.strip()
                ]

            if len(instruction_text_representations) > 0:
                function_text_representation = (
                    f"!{_maybe_translate_symbol(function_name, disassembly, logger)}:\n"
                    + "\n".join(instruction_text_representations)
                )

                function["TextRepresentation"] = function_text_representation
                function["Dependencies"] = dependencies
            else:
                del section["functions"][function_name]

    return disassembly


```

```
                            is not None
                        ):
                            offset = expression[expression_offset_index:]
                        else:
                            expression_offset_index = -1
                            offset = ""

                        symbol_end_index = (
                            expression_offset_index
                            if expression_offset_index != -1
                            else len(expression)
                        )

                        symbol = expression[:symbol_end_index]
                        dependencies += [_sanitize_symbol(symbol)]

                        symbol = _maybe_translate_symbol(symbol, disassembly, logger)

                        instruction_text_representation = (
                            instruction_text_representation.replace(
                                operand_in_question, f"{symbol}{offset}"
                            )
                        )

                instruction_text_representations += [
                    instruction_text_representation.strip()
                ]

            if len(instruction_text_representations) > 0:
                function_text_representation = (
                    f"!{_maybe_translate_symbol(function_name, disassembly, logger)}:\n"
                    + "\n".join(instruction_text_representations)
                )

                function["TextRepresentation"] = function_text_representation
                function["Dependencies"] = dependencies
            else:
                del section["functions"][function_name]

    return disassembly


def extract_examples_malware(testcase: dict,
                             label_granularity: typing.Optional[Granularity],
                             context_granularity: typing.Optional[Granularity],
                             emit_primary_good_function: bool,
```

```
                            logger: logging.Logger,
                            ) -> list[dict]:
    """Transform a testcase into a set of examples by disassembling and processing them."""
    # First, disassemble all objects
    object_files = [
        file
        for file in testcase["Files"]
        if file["Path"].name.endswith(".o")
           and not file["Path"].name.startswith("linked-")
    ]
    if not len(object_files) == 0:

        object_file = object_files[0]
        elf_path = object_file["Path"]

        disassembled_code = disassembler.disassemble_elf([elf_path], logger)
        disassembly = _preprocess_disassembly(disassembled_code, logger)

        examples = []
        if label_granularity == Granularity.FUNCTION:
            text_section_functions = disassembly["sections"][".text"]["functions"]

            # Decide which functions to use as examples
            primary_good_functions = []
            primary_bad_functions = []
            secondary_good_functions = []

            if str(disassembly['name']).endswith("_good.o"):
                primary_good_functions.append(disassembly['name'])
            else:
                primary_bad_functions.append(disassembly['name'])

            # Emit labeled functions
            emitted_functions = secondary_good_functions + primary_bad_functions
            if emit_primary_good_function:
                emitted_functions += primary_good_functions

            for function_name in emitted_functions:
                function = text_section_functions["_start"]
                text_representation = ""
                for text_section in text_section_functions:
                    text_representation += text_section_functions[text_section]["TextRepresentation"]
                example = {
                    "Example": text_representation,
                    "Testcase": testcase,
                    "GoodOrBad": "Good"
                    if function_name not in primary_bad_functions
                    else "Bad",
                }
                examples += [example]
```

```python
def extract_examples(
    testcase: dict,
    label_granularity: typing.Optional[Granularity],
    context_granularity: typing.Optional[Granularity],
    emit_primary_good_function: bool,
    logger: logging.Logger,
) -> list[dict]:
    """Transform a testcase into a set of examples by disassembling and processing them."""
    # First, disassemble all objects
    object_files = [
        file
        for file in testcase["Files"]
        if file["Path"].name.endswith(".o")
        and not file["Path"].name.startswith("linked-")
    ]

    disassembly = _preprocess_disassembly(
        disassembler.disassemble_elf(
            [object_file["Path"] for object_file in object_files], logger
        ),
        logger,
    )

    examples = []
    if label_granularity == Granularity.FUNCTION:
        text_section_functions = disassembly["sections"][".text"]["functions"]

        # Decide which functions to use as examples
        primary_good_functions = []
        primary_bad_functions = []
        secondary_good_functions = []

        for function_name in text_section_functions:
            if re.match(r"^(CWE.*(_|::))?bad(\(\))?$", function_name) is not None:
                primary_bad_functions += [function_name]
            elif re.match(r"^(CWE.*(_|::))?good(\(\))?$", function_name) is not None:
                primary_good_functions += [function_name]
            elif (
                re.match(
                    r"^(CWE.*(_|::))?good(\d+|G2B\d*|B2G\d*)(\(\))?$", function_name
                )
                is not None
            ):
```

```python

        else:
            raise ValueError("Granularity not supported")

        return examples
    else:
        print("No Object Files")
        exit(1)


def extract_examples(
    testcase: dict,
    label_granularity: typing.Optional[Granularity],
    context_granularity: typing.Optional[Granularity],
    emit_primary_good_function: bool,
    logger: logging.Logger,
) -> list[dict]:
    """Transform a testcase into a set of examples by disassembling and processing them."""
    # First, disassemble all objects
    object_files = [
        file
        for file in testcase["Files"]
        if file["Path"].name.endswith(".o")
        and not file["Path"].name.startswith("linked-")
    ]

    disassembly = _preprocess_disassembly(
        disassembler.disassemble_elf(
            [object_file["Path"] for object_file in object_files], logger
        ),
        logger,
    )

    examples = []
    if label_granularity == Granularity.FUNCTION:
        text_section_functions = disassembly["sections"][".text"]["functions"]

        # Decide which functions to use as examples
        primary_good_functions = []
        primary_bad_functions = []
        secondary_good_functions = []

        for function_name in text_section_functions:
            if re.match(r"^(CWE.*(_|::))?bad(\(\))?$", function_name) is not None:
                primary_bad_functions += [function_name]
            elif re.match(r"^(CWE.*(_|::))?good(\(\))?$", function_name) is not None:
                primary_good_functions += [function_name]
            elif (
                re.match(
                    r"^(CWE.*(_|::))?good(\d+|G2B\d*|B2G\d*)(\(\))?$", function_name
                )
                is not None
            ):
```

```
                secondary_good_functions += [function_name]

        # See if all functions are there

        # Ignore missing good functions for "Bad-Only Test Cases" (see Appendix D in Juliet 1.2 doc)
        if (
            testcase["Weakness"]["WeaknessID"] != 506
            or testcase["FunctionalVariant"]
            not in [
                "email",
                "file_transfer_connect_socket",
                "file_transfer_listen_socket",
                "screen_capture",
            ]
        ) and (
            testcase["Weakness"]["WeaknessID"] != 510
            or testcase["FunctionalVariant"]
            not in ["network_connection", "network_listen"]
        ):
            if not (len(primary_good_functions) > 0) and (
                len(secondary_good_functions) > 0
            ):
                logger.warning(
                    f"Number of primary or secondary good functions is zero for testcase {str(testcase)}!"
                )

        if not (len(primary_bad_functions) > 0):
            logger.warning(
                f"Number of primary bad functions is zero for testcase {str(testcase)}!"
            )

        # Emit labeled functions
        emitted_functions = secondary_good_functions + primary_bad_functions
        if emit_primary_good_function:
            emitted_functions += primary_good_functions

        for function_name in emitted_functions:
            function = text_section_functions[function_name]

            example = {
                "Example": function["TextRepresentation"],
                "Testcase": testcase,
                "GoodOrBad": "Good"
                if function_name not in primary_bad_functions
                else "Bad",
            }
            if context_granularity == Granularity.FUNCTION:
                pass
            elif context_granularity == Granularity.OBJECT:
```


```
                secondary_good_functions += [function_name]

        # See if all functions are there

        # Ignore missing good functions for "Bad-Only Test Cases" (see Appendix D in Juliet 1.2 doc)
        if (
            testcase["Weakness"]["WeaknessID"] != 506
            or testcase["FunctionalVariant"]
            not in [
                "email",
                "file_transfer_connect_socket",
                "file_transfer_listen_socket",
                "screen_capture",
            ]
        ) and (
            testcase["Weakness"]["WeaknessID"] != 510
            or testcase["FunctionalVariant"]
            not in ["network_connection", "network_listen"]
        ):
            if not (len(primary_good_functions) > 0) and (
                len(secondary_good_functions) > 0
            ):
                logger.warning(
                    f"Number of primary or secondary good functions is zero for testcase {str(testcase)}!"
                )

        if not (len(primary_bad_functions) > 0):
            logger.warning(
                f"Number of primary bad functions is zero for testcase {str(testcase)}!"
            )

        # Emit labeled functions
        emitted_functions = secondary_good_functions + primary_bad_functions
        if emit_primary_good_function:
            emitted_functions += primary_good_functions

        for function_name in emitted_functions:
            function = text_section_functions[function_name]

            example = {
                "Example": function["TextRepresentation"],
                "Testcase": testcase,
                "GoodOrBad": "Good"
                if function_name not in primary_bad_functions
                else "Bad",
            }
            if context_granularity == Granularity.FUNCTION:
                pass
            elif context_granularity == Granularity.OBJECT:
```

```
                # Ignore functions that are not helpful or leak labels
                ignored_functions = [
                    "_start",
                    "__libc_csu_init",
                    "__libc_csu_fini",
                    "_dl_relocate_static_pie",
                    "deregister_tm_clones",
                    "register_tm_clones",
                    "__do_global_dtors_aux",
                    "frame_dummy",
                ] + [function_name]

                if function_name in primary_bad_functions:
                    ignored_functions += (
                        primary_good_functions + secondary_good_functions
                    )
                else:
                    if not (
                        function_name in primary_good_functions
                        or function_name in secondary_good_functions
                    ):
                        logger.warning(
                            f"Function {function_name} not found in primary or secondary good functions or bad functions for testcase {str(testcase)}!"
                        )
                        logger.warning(
                            f"Candidate functions: {primary_good_functions}+{secondary_good_functions}+{primary_bad_functions}"
                        )
                    ignored_functions += primary_bad_functions

                # Build list of candidate functions
                candidate_functions = {function_name}
                while True:
                    new_candidate_functions = set(candidate_functions)
                    for candidate_function in candidate_functions:
                        for dependency in text_section_functions[candidate_function][
                            "Dependencies"
                        ]:
                            if dependency in text_section_functions.keys():
                                new_candidate_functions.add(dependency)
                    if len(new_candidate_functions) == len(candidate_functions):
                        break
                    else:
                        candidate_functions = new_candidate_functions

                context_functions = [
                    fnn for fnn in candidate_functions if fnn not in ignored_functions
                ]
```

```
                # Ignore functions that are not helpful or leak labels
                ignored_functions = [
                    "_start",
                    "__libc_csu_init",
                    "__libc_csu_fini",
                    "_dl_relocate_static_pie",
                    "deregister_tm_clones",
                    "register_tm_clones",
                    "__do_global_dtors_aux",
                    "frame_dummy",
                ] + [function_name]

                if function_name in primary_bad_functions:
                    ignored_functions += (
                        primary_good_functions + secondary_good_functions
                    )
                else:
                    if not (
                        function_name in primary_good_functions
                        or function_name in secondary_good_functions
                    ):
                        logger.warning(
                            f"Function {function_name} not found in primary or secondary good functions or bad functions for testcase {str(testcase)}!"
                        )
                        logger.warning(
                            f"Candidate functions: {primary_good_functions}+{secondary_good_functions}+{primary_bad_functions}"
                        )
                    ignored_functions += primary_bad_functions

                # Build list of candidate functions
                candidate_functions = {function_name}
                while True:
                    new_candidate_functions = set(candidate_functions)
                    for candidate_function in candidate_functions:
                        for dependency in text_section_functions[candidate_function][
                            "Dependencies"
                        ]:
                            if dependency in text_section_functions.keys():
                                new_candidate_functions.add(dependency)
                    if len(new_candidate_functions) == len(candidate_functions):
                        break
                    else:
                        candidate_functions = new_candidate_functions

                context_functions = [
                    fnn for fnn in candidate_functions if fnn not in ignored_functions
                ]
```

```
                context = "\n".join(
                    [
                        text_section_functions[fnn]["TextRepresentation"]
                        for fnn in context_functions
                    ]
                )

                example["Example"] += "\n" + context
            elif context_granularity is None:
                pass
            else:
                raise ValueError("Granularity not supported for context")

            examples += [example]

    else:
        raise ValueError("Granularity not supported")

    return examples


def label_example(example, label_strategy: LabelStrategy):
    """Compute a label for a given example depending on the label strategy."""
    if label_strategy == LabelStrategy.BINARYCLASSIFICATION:
        return example["GoodOrBad"]
    elif label_strategy == LabelStrategy.MULTICLASSCLASSIFICATION:
        if example["GoodOrBad"] == "Good":
            return "NoWeakness"
        else:
            return f"CWE{example['Testcase']['Weakness']['WeaknessID']}"

    else:
        raise ValueError("Label strategy not supported")


__all__ = ["extract_labeled_examples", "validate_testcase_files", "label_example"]
```

```
                context = "\n".join(
                    [
                        text_section_functions[fnn]["TextRepresentation"]
                        for fnn in context_functions
                    ]
                )

                example["Example"] += "\n" + context
            elif context_granularity is None:
                pass
            else:
                raise ValueError("Granularity not supported for context")

            examples += [example]

    else:
        raise ValueError("Granularity not supported")

    return examples


def label_example(example, label_strategy: LabelStrategy):
    """Compute a label for a given example depending on the label strategy."""
    if label_strategy == LabelStrategy.BINARYCLASSIFICATION:
        return example["GoodOrBad"]
    elif label_strategy == LabelStrategy.MULTICLASSCLASSIFICATION:
        if example["GoodOrBad"] == "Good":
            return "NoWeakness"
        else:
            return f"CWE{example['Testcase']['Weakness']['WeaknessID']}"

    else:
        raise ValueError("Label strategy not supported")


__all__ = ["extract_labeled_examples", "validate_testcase_files", "label_example"]
```