■ Code Analysis Results

===== String Encryption =====

input\controlFlow.py: [0] -> * (XOR-encrypted string detected and decrypted)
input\controlFlow.py: [0] -> * (XOR-encrypted string detected and decrypted)
input\deadCode.py: [0] -> * (XOR-encrypted string detected and decrypted)
input\deadCode.py: [0] -> * (XOR-encrypted string detected and decrypted)
input\nameIdentifier.py: {1,3} -> +) (XOR-encrypted string detected and decrypted)
input\opaque_predicate.py: [4, 5] -> ./ (XOR-encrypted string detected and decrypted)
input\opaque_predicate.py: [0] -> * (XOR-encrypted string detected and decrypted)
input\opaque_predicate.py: [0] -> * (XOR-encrypted string detected and decrypted)

===== Identifier Cleaner =====

input\junkcode.py: m1 -> var1 (Obfuscated identifier replaced (m1 -> var1))

input\junkcode.py: m2 -> var2 (Obfuscated identifier replaced (m2 -> var2))

input\junkcode.py: m1 -> var1 (Obfuscated identifier replaced (m1 -> var1))

input\junkcode.py: m2 -> var2 (Obfuscated identifier replaced (m2 -> var2))

input\junkcode.py: m1 -> var1 (Obfuscated identifier replaced (m1 -> var1))

input\junkcode.py: m2 -> var2 (Obfuscated identifier replaced (m2 -> var2))

input\junkcode.py: m1 -> var1 (Obfuscated identifier replaced (m1 -> var1))

input\junkcode.py: m2 -> var2 (Obfuscated identifier replaced (m2 -> var2))

input\junkcode.py: m1 -> var1 (Obfuscated identifier replaced (m1 -> var1))

input\junkcode.py: m2 -> var2 (Obfuscated identifier replaced (m2 -> var2))

input\junkcode.py: m1 -> var1 (Obfuscated identifier replaced (m1 -> var1))

input\junkcode.py: m2 -> var2 (Obfuscated identifier replaced (m2 -> var2))

===== Control Flow =====

-> (Unreachable branch (condition always False))

-> (Unreachable branch (condition always False))

-> print("Always runs")

-> print("Run")

input\deadCode.py: if(false) { ... }` blocks -> (Unreachable branch (condition always false))

input\deadCode.py: if(false) { ... } or if(false) statement; -> (Unreachable branch (condition always false))

input\deadCode.py: if(false) removed (dead code)"}) -> (Unreachable branch (condition always false))

input\deadCode.py: if(true) { ... }` with block contents (strip braces) -> { ... }` with block contents (strip braces) (Always-true condition simplified (kept statement/block, removed condition header))

input\deadCode.py: if(true) { block } => replace with block contents -> { block } => replace with block contents (Always-true condition simplified (kept statement/block, removed condition header))

input\deadCode.py: if(true) inlined (kept body)"}) -> inlined (kept body)"}) (Always-true condition simplified (kept statement/block, removed condition header))

===== Dead Code =====

def detect_api_redirection_clike(code: str) -> List[Dict]:

def clean_api_redirection_clike(code: str) -> List[Dict]:

changes.append({"original": call_name + "()", "cleaned": actual + "()", "reason": f"Inlined trivial wrapper {call_name} -> {actual}"})

def detect_api_redirection_python(code: str) -> List[Dict]:

def clean_api_redirection_python(code: str) -> List[Dict]:

changes.append({"original": wrapper + "() calls", "cleaned": target + "()", "reason": f"Inlined trivial wrapper {wrapper} -> {target}"})

-> """

def detect_api_redirection_clike(code: str) -> List[Dict]:

def clean_api_redirection_clike(code: str) -> List[Dict]:

changes.append({'original': call_name + '()', 'cleaned': actual + '()', 'reason': f'Inlined trivial wrapper {call_name} -> {actual}'})

def detect_api_redirection_python(code: str) -> List[Dict]:

def clean_api_redirection_python(code: str) -> List[Dict]:

changes.append({'original': wrapper + '() calls', 'cleaned': target + '()', 'reason': f'Inlined trivial wrapper {wrapper} -> {target}'})

def _extract_python_if_block(self, code: str, match_start: int) -> Tuple[str, str]:

def _dedent_python_body(self, body_text: str) -> str:

def _extract_c_like_block_or_line(self, code: str, start_idx: int) -> Tuple[str, str]:

def detect_fake_conditions(self, code: str) -> List[Dict]:

def clean_code(self, code: str) -> List[Dict]:

return changes -> # ---------- controlflow.py ----------

def _extract_python_if_block(self, code: str, match_start: int) -> Tuple[str, str]:

def _dedent_python_body(self, body_text: str) -> str:

def _extract_c_like_block_or_line(self, code: str, start_idx: int) -> Tuple[str, str]:

def detect_fake_conditions(self, code: str) -> List[Dict]:

def clean_code(self, code: str) -> List[Dict]:

def detect_controlflow_flattening_clike(code: str) -> List[Dict]:

def clean_controlflow_flattening_clike(code: str) -> List[Dict]:

def detect_controlflow_flattening_python(code: str) -> List[Dict]:

def clean_controlflow_flattening_python(code: str) -> List[Dict]:

-> """

def detect_controlflow_flattening_clike(code: str) -> List[Dict]:

def clean_controlflow_flattening_clike(code: str) -> List[Dict]:

def detect_controlflow_flattening_python(code: str) -> List[Dict]:

def clean_controlflow_flattening_python(code: str) -> List[Dict]:

def detect_deadcode_python(code: str) -> List[Dict]:

def clean_deadcode_python(code: str) -> List[Dict]:

def detect_deadcode_clike(code: str, ext_tag: str = "C-like") -> List[Dict]:

def clean_deadcode_clike(code: str) -> List[Dict]:

-> """# Dead Code Test Cases

def detect_deadcode_python(code: str) -> List[Dict]:

def clean_deadcode_python(code: str) -> List[Dict]:

def detect_deadcode_clike(code: str, ext_tag: str = "C-like") -> List[Dict]:

def clean_deadcode_clike(code: str) -> List[Dict]:

Cleaning is conservative: we only report and, for trivial indirect calls (wrapper->function pointer),

def detect_dynamic_code_loading_python(code: str) -> List[Dict]:

def detect_dynamic_code_loading_clike(code: str) -> List[Dict]:

def clean_dynamic_code_loading_clike(code: str) -> List[Dict]:

changes.append({"original": orig_wrapper, "cleaned": "", "reason": f"Removed trivial wrapper {wrapper_name} -> inlined calls to {real}()"})

def clean_dynamic_code_loading_python(code: str) -> List[Dict]:

-> """

Cleaning is conservative: we only report and, for trivial indirect calls (wrapper->function pointer),

def detect_dynamic_code_loading_python(code: str) -> List[Dict]:

def detect_dynamic_code_loading_clike(code: str) -> List[Dict]:

def clean_dynamic_code_loading_clike(code: str) -> List[Dict]:

changes.append({'original': orig_wrapper, 'cleaned': '', 'reason': f'Removed trivial wrapper {wrapper_name} -> inlined calls to {real}()'})

def clean_dynamic_code_loading_python(code: str) -> List[Dict]:

raise ValueError("Not a constant-evaluable expression")

```
def detect_inline_expansion_python(code: str) -> List[Dict]:
def clean_inline_expansion_python(code: str) -> List[Dict]:
def detect_inline_expansion_clike(code: str, ext_tag="C-like") -> List[Dict]:
def clean_inline_expansion_clike(code: str) -> List[Dict]:
if b_i == 0: raise ZeroDivisionError
raise ValueError
-> """# Inline Expansion Complex Test Cases with Loops
raise ValueError('Not a constant-evaluable expression')
def detect_inline_expansion_python(code: str) -> List[Dict]:
def clean_inline_expansion_python(code: str) -> List[Dict]:
def detect_inline_expansion_clike(code: str, ext_tag='C-like') -> List[Dict]:
def clean_inline_expansion_clike(code: str) -> List[Dict]:
raise ZeroDivisionError
raise ValueError
- x - (-1) -> x + 1
- x << 1 -> x * 2
- x + x -> 2 * x (or x * 2)
# Patterns -> canonical replacement
# x - (-1) -> x + 1
# x << 1 -> x * 2
# x + x -> 2 * x (note: keep order)
# double-negation: --x -> x (in C/C++)
def detect_instruction_substitution_clike(code: str) -> List[Dict]:
def clean_instruction_substitution_clike(code: str) -> List[Dict]:
def detect_instruction_substitution_python(code: str) -> List[Dict]:
def clean_instruction_substitution_python(code: str) -> List[Dict]:
-> """
- x - (-1) -> x + 1
- x << 1 -> x * 2
- x + x -> 2 * x (or x * 2)
def detect_instruction_substitution_clike(code: str) -> List[Dict]:
def clean_instruction_substitution_clike(code: str) -> List[Dict]:
```

```
def detect_instruction_substitution_python(code: str) -> List[Dict]:
def clean_instruction_substitution_python(code: str) -> List[Dict]:
def detect_junk_code_clike(code: str) -> List[Dict]:
def clean_junk_code_clike(code: str) -> List[Dict]:
def detect_junk_code_python(code: str) -> List[Dict]:
def clean_junk_code_python(code: str) -> List[Dict]:
-> """
def detect_junk_code_clike(code: str) -> List[Dict]:
def clean_junk_code_clike(code: str) -> List[Dict]:
def detect_junk_code_python(code: str) -> List[Dict]:
def clean_junk_code_python(code: str) -> List[Dict]:
def detect_mixed_language(code: str) -> List[Dict]:
def clean_mixed_language_python(code: str) -> List[Dict]:
def clean_mixed_language_clike(code: str) -> List[Dict]:
-> """
def detect_mixed_language(code: str) -> List[Dict]:
def clean_mixed_language_python(code: str) -> List[Dict]:
def clean_mixed_language_clike(code: str) -> List[Dict]:
def is_obfuscated_name(name: str, language: str = "python") -> bool:
def detect_language(filename: str) -> str:
return "unknown" -> import re
def is_obfuscated_name(name: str, language: str = "python") -> bool:
def detect_language(filename: str) -> str:
raise ValueError("Not constant-evaluable")
def detect_opaque_predicate_python(code: str) -> List[Dict]:
def clean_opaque_predicate_python(code: str) -> List[Dict]:
self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated True -> inlined body"})
self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated False -> removed or replaced with else"})
raise ValueError("unsafe expr")
def detect_opaque_predicate_clike(code: str, lang="C-like") -> List[Dict]:
```

def clean_opaque_predicate_clike(code: str) -> List[Dict]:

- if condition is always true: replace `if(cond){block}else{else}` -> keep block

-> # Opaque Predicate Complex Test Cases

raise ValueError("Not constant-evaluable")

def detect_opaque_predicate_python(code: str) -> List[Dict]:

def clean_opaque_predicate_python(code: str) -> List[Dict]:

self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated True -> inlined body"})

self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated False -> removed or replaced with else"})

raise ValueError("unsafe expr")

def detect_opaque_predicate_clike(code: str, lang="C-like") -> List[Dict]:

def clean_opaque_predicate_clike(code: str) -> List[Dict]:

- if condition is always true: replace `if(cond){block}else{else}` -> keep block

return results, cleaned_code -> import re

===== Inline Expansion =====

def detect_api_redirection_clike(code: str) -> List[Dict]:

def clean_api_redirection_clike(code: str) -> List[Dict]:

changes.append({'original': call_name + '()', 'cleaned': actual + '()', 'reason': f'Inlined trivial wrapper {call_name} -> {actual}'})

def detect_api_redirection_python(code: str) -> List[Dict]:

def clean_api_redirection_python(code: str) -> List[Dict]:

changes.append({'original': wrapper + '() calls', 'cleaned': target + '()', 'reason': f'Inlined trivial wrapper {wrapper} -> {target}'})

return changes -> """

def detect_api_redirection_clike(code: str) -> List[Dict]:

def clean_api_redirection_clike(code: str) -> List[Dict]:

changes.append({'original': call_name + '()', 'cleaned': actual + '()', 'reason': f'Inlined trivial wrapper {call_name} -> {actual}'})

def detect_api_redirection_python(code: str) -> List[Dict]:

def clean_api_redirection_python(code: str) -> List[Dict]:

changes.append({'original': wrapper + '() calls', 'cleaned': target + '()', 'reason': f'Inlined trivial wrapper {wrapper} -> {target}'})

def _extract_python_if_block(self, code: str, match_start: int) -> Tuple[str, str]:

def _dedent_python_body(self, body_text: str) -> str:

def _extract_c_like_block_or_line(self, code: str, start_idx: int) -> Tuple[str, str]:

def detect_fake_conditions(self, code: str) -> List[Dict]:

def clean_code(self, code: str) -> List[Dict]:

return changes -> # ---------- controlflow.py ----------

def _extract_python_if_block(self, code: str, match_start: int) -> Tuple[str, str]:

def _dedent_python_body(self, body_text: str) -> str:

def _extract_c_like_block_or_line(self, code: str, start_idx: int) -> Tuple[str, str]:

def detect_fake_conditions(self, code: str) -> List[Dict]:

def clean_code(self, code: str) -> List[Dict]:

input\controlflow_flattening.py: '// Suggested deobfuscated sequence\n' + '/* case1 */\n' -> '// Suggested deobfuscated sequence\n/* case1 */\n' (Constant folded)

def detect_controlflow_flattening_clike(code: str) -> List[Dict]:

def clean_controlflow_flattening_clike(code: str) -> List[Dict]:

def detect_controlflow_flattening_python(code: str) -> List[Dict]:

def clean_controlflow_flattening_python(code: str) -> List[Dict]:

return [] -> """

def detect_controlflow_flattening_clike(code: str) -> List[Dict]:

def clean_controlflow_flattening_clike(code: str) -> List[Dict]:

def detect_controlflow_flattening_python(code: str) -> List[Dict]:

def clean_controlflow_flattening_python(code: str) -> List[Dict]:

def detect_deadcode_python(code: str) -> List[Dict]:

def clean_deadcode_python(code: str) -> List[Dict]:

def detect_deadcode_clike(code: str, ext_tag: str = "C-like") -> List[Dict]:

def clean_deadcode_clike(code: str) -> List[Dict]:

-> """# Dead Code Test Cases

def detect_deadcode_python(code: str) -> List[Dict]:

def clean_deadcode_python(code: str) -> List[Dict]:

def detect_deadcode_clike(code: str, ext_tag: str = "C-like") -> List[Dict]:

def clean_deadcode_clike(code: str) -> List[Dict]:

Cleaning is conservative: we only report and, for trivial indirect calls (wrapper->function pointer),

def detect_dynamic_code_loading_python(code: str) -> List[Dict]:

def detect_dynamic_code_loading_clike(code: str) -> List[Dict]:

def clean_dynamic_code_loading_clike(code: str) -> List[Dict]:

changes.append({'original': orig_wrapper, 'cleaned': '', 'reason': f'Removed trivial wrapper {wrapper_name} -> inlined calls to {real}()'})

def clean_dynamic_code_loading_python(code: str) -> List[Dict]:

return [] -> """

Cleaning is conservative: we only report and, for trivial indirect calls (wrapper->function pointer),

def detect_dynamic_code_loading_python(code: str) -> List[Dict]:

def detect_dynamic_code_loading_clike(code: str) -> List[Dict]:

def clean_dynamic_code_loading_clike(code: str) -> List[Dict]:

changes.append({'original': orig_wrapper, 'cleaned': '', 'reason': f'Removed trivial wrapper {wrapper_name} -> inlined calls to {real}()'})

```
def clean_dynamic_code_loading_python(code: str) -> List[Dict]:
raise ValueError('Not a constant-evaluable expression')
def detect_inline_expansion_python(code: str) -> List[Dict]:
def clean_inline_expansion_python(code: str) -> List[Dict]:
def detect_inline_expansion_clike(code: str, ext_tag='C-like') -> List[Dict]:
def clean_inline_expansion_clike(code: str) -> List[Dict]:
raise ZeroDivisionError
raise ValueError
return changes -> """# Inline Expansion Complex Test Cases with Loops
raise ValueError('Not a constant-evaluable expression')
def detect_inline_expansion_python(code: str) -> List[Dict]:
def clean_inline_expansion_python(code: str) -> List[Dict]:
def detect_inline_expansion_clike(code: str, ext_tag='C-like') -> List[Dict]:
def clean_inline_expansion_clike(code: str) -> List[Dict]:
raise ZeroDivisionError
raise ValueError
- x - (-1) -> x + 1
- x << 1 -> x * 2
- x + x -> 2 * x (or x * 2)
def detect_instruction_substitution_clike(code: str) -> List[Dict]:
def clean_instruction_substitution_clike(code: str) -> List[Dict]:
def detect_instruction_substitution_python(code: str) -> List[Dict]:
def clean_instruction_substitution_python(code: str) -> List[Dict]:
return changes -> """
- x - (-1) -> x + 1
- x << 1 -> x * 2
- x + x -> 2 * x (or x * 2)
def detect_instruction_substitution_clike(code: str) -> List[Dict]:
def clean_instruction_substitution_clike(code: str) -> List[Dict]:
def detect_instruction_substitution_python(code: str) -> List[Dict]:
def clean_instruction_substitution_python(code: str) -> List[Dict]:
def detect_junk_code_clike(code: str) -> List[Dict]:
```

def clean_junk_code_clike(code: str) -> List[Dict]:

def detect_junk_code_python(code: str) -> List[Dict]:

def clean_junk_code_python(code: str) -> List[Dict]:

return changes -> """

def detect_junk_code_clike(code: str) -> List[Dict]:

def clean_junk_code_clike(code: str) -> List[Dict]:

def detect_junk_code_python(code: str) -> List[Dict]:

def clean_junk_code_python(code: str) -> List[Dict]:

def detect_mixed_language(code: str) -> List[Dict]:

def clean_mixed_language_python(code: str) -> List[Dict]:

def clean_mixed_language_clike(code: str) -> List[Dict]:

return changes -> """

def detect_mixed_language(code: str) -> List[Dict]:

def clean_mixed_language_python(code: str) -> List[Dict]:

def clean_mixed_language_clike(code: str) -> List[Dict]:

def is_obfuscated_name(name: str, language: str = "python") -> bool:

def detect_language(filename: str) -> str:

return "unknown" -> import re

def is_obfuscated_name(name: str, language: str = "python") -> bool:

def detect_language(filename: str) -> str:

raise ValueError("Not constant-evaluable")

def detect_opaque_predicate_python(code: str) -> List[Dict]:

def clean_opaque_predicate_python(code: str) -> List[Dict]:

self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated True -> inlined body"})

self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated False -> removed or replaced with else"})

raise ValueError("unsafe expr")

def detect_opaque_predicate_clike(code: str, lang="C-like") -> List[Dict]:

def clean_opaque_predicate_clike(code: str) -> List[Dict]:

- if condition is always true: replace `if(cond){block}else{else}` -> keep block

-> # Opaque Predicate Complex Test Cases

raise ValueError("Not constant-evaluable")

def detect_opaque_predicate_python(code: str) -> List[Dict]:

def clean_opaque_predicate_python(code: str) -> List[Dict]:

self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated True -> inlined body"})

self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated False -> removed or replaced with else"})

raise ValueError("unsafe expr")

def detect_opaque_predicate_clike(code: str, lang="C-like") -> List[Dict]:

def clean_opaque_predicate_clike(code: str) -> List[Dict]:

- if condition is always true: replace `if(cond){block}else{else}` -> keep block

return (results, cleaned_code) -> import re

===== Opaque Predicates =====

def detect_api_redirection_clike(code: str) -> List[Dict]:

def clean_api_redirection_clike(code: str) -> List[Dict]:

changes.append({'original': call_name + '()', 'cleaned': actual + '()', 'reason': f'Inlined trivial wrapper {call_name} -> {actual}'})

def detect_api_redirection_python(code: str) -> List[Dict]:

def clean_api_redirection_python(code: str) -> List[Dict]:

changes.append({'original': wrapper + '() calls', 'cleaned': target + '()', 'reason': f'Inlined trivial wrapper {wrapper} -> {target}'})

return changes -> """

def detect_api_redirection_clike(code: str) -> List[Dict]:

def clean_api_redirection_clike(code: str) -> List[Dict]:

changes.append({'original': call_name + '()', 'cleaned': actual + '()', 'reason': f'Inlined trivial wrapper {call_name} -> {actual}'})

def detect_api_redirection_python(code: str) -> List[Dict]:

def clean_api_redirection_python(code: str) -> List[Dict]:

changes.append({'original': wrapper + '() calls', 'cleaned': target + '()', 'reason': f'Inlined trivial wrapper {wrapper} -> {target}'})

def _extract_python_if_block(self, code: str, match_start: int) -> Tuple[str, str]:

def _dedent_python_body(self, body_text: str) -> str:

def _extract_c_like_block_or_line(self, code: str, start_idx: int) -> Tuple[str, str]:

def detect_fake_conditions(self, code: str) -> List[Dict]:

def clean_code(self, code: str) -> List[Dict]:

return changes -> # ---------- controlflow.py ----------

def _extract_python_if_block(self, code: str, match_start: int) -> Tuple[str, str]:

def _dedent_python_body(self, body_text: str) -> str:

def _extract_c_like_block_or_line(self, code: str, start_idx: int) -> Tuple[str, str]:

def detect_fake_conditions(self, code: str) -> List[Dict]:

def clean_code(self, code: str) -> List[Dict]:

def detect_controlflow_flattening_clike(code: str) -> List[Dict]:

def clean_controlflow_flattening_clike(code: str) -> List[Dict]:

def detect_controlflow_flattening_python(code: str) -> List[Dict]:

def clean_controlflow_flattening_python(code: str) -> List[Dict]:

return [] -> """

def detect_controlflow_flattening_clike(code: str) -> List[Dict]:

def clean_controlflow_flattening_clike(code: str) -> List[Dict]:

def detect_controlflow_flattening_python(code: str) -> List[Dict]:

def clean_controlflow_flattening_python(code: str) -> List[Dict]:

def detect_deadcode_python(code: str) -> List[Dict]:

def clean_deadcode_python(code: str) -> List[Dict]:

def detect_deadcode_clike(code: str, ext_tag: str = "C-like") -> List[Dict]:

def clean_deadcode_clike(code: str) -> List[Dict]:

-> """# Dead Code Test Cases

def detect_deadcode_python(code: str) -> List[Dict]:

def clean_deadcode_python(code: str) -> List[Dict]:

def detect_deadcode_clike(code: str, ext_tag: str = "C-like") -> List[Dict]:

def clean_deadcode_clike(code: str) -> List[Dict]:

Cleaning is conservative: we only report and, for trivial indirect calls (wrapper->function pointer),

def detect_dynamic_code_loading_python(code: str) -> List[Dict]:

def detect_dynamic_code_loading_clike(code: str) -> List[Dict]:

def clean_dynamic_code_loading_clike(code: str) -> List[Dict]:

changes.append({'original': orig_wrapper, 'cleaned': '', 'reason': f'Removed trivial wrapper {wrapper_name} -> inlined calls to {real}()'})

def clean_dynamic_code_loading_python(code: str) -> List[Dict]:

return [] -> """

Cleaning is conservative: we only report and, for trivial indirect calls (wrapper->function pointer),

def detect_dynamic_code_loading_python(code: str) -> List[Dict]:

def detect_dynamic_code_loading_clike(code: str) -> List[Dict]:

def clean_dynamic_code_loading_clike(code: str) -> List[Dict]:

changes.append({'original': orig_wrapper, 'cleaned': '', 'reason': f'Removed trivial wrapper {wrapper_name} -> inlined calls to {real}()'})

def clean_dynamic_code_loading_python(code: str) -> List[Dict]:

raise ValueError('Not a constant-evaluable expression')

```
def detect_inline_expansion_python(code: str) -> List[Dict]:
def clean_inline_expansion_python(code: str) -> List[Dict]:
def detect_inline_expansion_clike(code: str, ext_tag='C-like') -> List[Dict]:
def clean_inline_expansion_clike(code: str) -> List[Dict]:
raise ZeroDivisionError
raise ValueError
return changes -> """# Inline Expansion Complex Test Cases with Loops
raise ValueError('Not a constant-evaluable expression')
def detect_inline_expansion_python(code: str) -> List[Dict]:
def clean_inline_expansion_python(code: str) -> List[Dict]:
def detect_inline_expansion_clike(code: str, ext_tag='C-like') -> List[Dict]:
def clean_inline_expansion_clike(code: str) -> List[Dict]:
raise ZeroDivisionError
raise ValueError
- x - (-1) -> x + 1
- x << 1 -> x * 2
- x + x -> 2 * x (or x * 2)
def detect_instruction_substitution_clike(code: str) -> List[Dict]:
def clean_instruction_substitution_clike(code: str) -> List[Dict]:
def detect_instruction_substitution_python(code: str) -> List[Dict]:
def clean_instruction_substitution_python(code: str) -> List[Dict]:
return changes -> """
- x - (-1) -> x + 1
- x << 1 -> x * 2
- x + x -> 2 * x (or x * 2)
def detect_instruction_substitution_clike(code: str) -> List[Dict]:
def clean_instruction_substitution_clike(code: str) -> List[Dict]:
def detect_instruction_substitution_python(code: str) -> List[Dict]:
def clean_instruction_substitution_python(code: str) -> List[Dict]:
def detect_junk_code_clike(code: str) -> List[Dict]:
def clean_junk_code_clike(code: str) -> List[Dict]:
def detect_junk_code_python(code: str) -> List[Dict]:
```

def clean_junk_code_python(code: str) -> List[Dict]:

return changes -> """

def detect_junk_code_clike(code: str) -> List[Dict]:

def clean_junk_code_clike(code: str) -> List[Dict]:

def detect_junk_code_python(code: str) -> List[Dict]:

def clean_junk_code_python(code: str) -> List[Dict]:

def detect_mixed_language(code: str) -> List[Dict]:

def clean_mixed_language_python(code: str) -> List[Dict]:

def clean_mixed_language_clike(code: str) -> List[Dict]:

return changes -> """

def detect_mixed_language(code: str) -> List[Dict]:

def clean_mixed_language_python(code: str) -> List[Dict]:

def clean_mixed_language_clike(code: str) -> List[Dict]:

def is_obfuscated_name(name: str, language: str = "python") -> bool:

def detect_language(filename: str) -> str:

return "unknown" -> import re

def is_obfuscated_name(name: str, language: str = "python") -> bool:

def detect_language(filename: str) -> str:

raise ValueError("Not constant-evaluable")

def detect_opaque_predicate_python(code: str) -> List[Dict]:

def clean_opaque_predicate_python(code: str) -> List[Dict]:

self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated True -> inlined body"})

self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated False -> removed or replaced with else"})

raise ValueError("unsafe expr")

def detect_opaque_predicate_clike(code: str, lang="C-like") -> List[Dict]:

def clean_opaque_predicate_clike(code: str) -> List[Dict]:

- if condition is always true: replace `if(cond){block}else{else}` -> keep block

-> # Opaque Predicate Complex Test Cases

raise ValueError("Not constant-evaluable")

def detect_opaque_predicate_python(code: str) -> List[Dict]:

def clean_opaque_predicate_python(code: str) -> List[Dict]:

self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated True -> inlined body"})

self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated False -> removed or replaced with else"})

raise ValueError("unsafe expr")

def detect_opaque_predicate_clike(code: str, lang="C-like") -> List[Dict]:

def clean_opaque_predicate_clike(code: str) -> List[Dict]:

- if condition is always true: replace `if(cond){block}else{else}` -> keep block

return (results, cleaned_code) -> import re

===== Control Flow Flattening =====

input\controlflow_flattening.py: -> (Detected Python flattened control-flow patterns. Manual reconstruction recommended.)

input\inlineExpansion.py: -> (Detected Python flattened control-flow patterns. Manual reconstruction recommended.)

===== Instruction Substitution =====

input\instruction_substitution.py: x - (-1) -> x + 1 (Canonicalized: neg-neg to plus)

input\instruction_substitution.py: x + x -> 2 * x (Canonicalized: x+x to 2*x)

input\instruction_substitution.py: x+x -> 2 * x (Canonicalized: x+x to 2*x)

input\instruction_substitution.py: x+x -> 2 * x (Canonicalized: x+x to 2*x)

- x - (-1) -> x + 1

- x << 1 -> x * 2

- x + x -> 2 * x (or x * 2)

def detect_instruction_substitution_clike(code: str) -> List[Dict]:

def clean_instruction_substitution_clike(code: str) -> List[Dict]:

def detect_instruction_substitution_python(code: str) -> List[Dict]:

def clean_instruction_substitution_python(code: str) -> List[Dict]:

return changes -> """

- x + 1 -> x + 1

- x << 1 -> x * 2

- 2 * x -> 2 * x (or x * 2)

def detect_instruction_substitution_clike(code: str) -> List[Dict]:

def clean_instruction_substitution_clike(code: str) -> List[Dict]:

def detect_instruction_substitution_python(code: str) -> List[Dict]:

def clean_instruction_substitution_python(code: str) -> List[Dict]:

===== Dynamic Code Loading =====

input\controlFlow.py: -> (Detected dynamic constructs (eval/exec/compile/reflection). Manual review required: [{'type': 'dynamic_py', 'lineno': 17, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 18, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 19, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 24, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 25, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 26, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}])

input\controlflow_flattening.py: -> (Detected dynamic constructs (eval/exec/compile/reflection). Manual review required: [{'type': 'dynamic_py', 'lineno': 11, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}])

input\deadCode.py: -> (Detected dynamic constructs (eval/exec/compile/reflection). Manual review required: [{'type': 'dynamic_py', 'lineno': 228, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 237, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 252, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 129, 'snippet': 'getattr(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 144, 'snippet': 'getattr(', 'reason': 'dynamic execution/reflection'}])

input\dynamic_loading.py: -> (Detected dynamic constructs (eval/exec/compile/reflection). Manual review required: [{'type': 'dynamic_py', 'lineno': 11, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 11, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 11, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 11, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 11, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 11, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}])

input\inlineExpansion.py: -> (Detected dynamic constructs (eval/exec/compile/reflection). Manual review required: [{'type': 'dynamic_py', 'lineno': 153, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}])

input\instruction_substitution.py: -> (Detected dynamic constructs (eval/exec/compile/reflection). Manual review required: [{'type': 'dynamic_py', 'lineno': 11, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 11, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 11, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 11, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 32, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 32, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}])

input\opaque_predicate.py: -> (Detected dynamic constructs (eval/exec/compile/reflection). Manual review required: [{'type': 'dynamic_py', 'lineno': 198, 'snippet': 'eval(', 'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 189, 'snippet': 'compile(',

'reason': 'dynamic execution/reflection'}, {'type': 'dynamic_py', 'lineno': 231, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}])

input\stringEncryption.py: -> (Detected dynamic constructs (eval/exec/compile/reflection). Manual review required: [{'type': 'dynamic_py', 'lineno': 18, 'snippet': 'compile(', 'reason': 'dynamic execution/reflection'}])

===== Junk Code =====

def detect_deadcode_python(code: str) -> List[Dict]:

def clean_deadcode_python(code: str) -> List[Dict]:

def detect_deadcode_clike(code: str, ext_tag: str = "C-like") -> List[Dict]:

def clean_deadcode_clike(code: str) -> List[Dict]:

-> """# Dead Code Test Cases

def detect_deadcode_python(code: str) -> List[Dict]:

def clean_deadcode_python(code: str) -> List[Dict]:

def detect_deadcode_clike(code: str, ext_tag: str = "C-like") -> List[Dict]:

def clean_deadcode_clike(code: str) -> List[Dict]:

def detect_junk_code_clike(code: str) -> List[Dict]:

def clean_junk_code_clike(code: str) -> List[Dict]:

def detect_junk_code_python(code: str) -> List[Dict]:

def clean_junk_code_python(code: str) -> List[Dict]:

return changes -> """

def detect_junk_code_clike(code: str) -> List[Dict]:

def clean_junk_code_clike(code: str) -> List[Dict]:

def detect_junk_code_python(code: str) -> List[Dict]:

def clean_junk_code_python(code: str) -> List[Dict]:

raise ValueError("Not constant-evaluable")

def detect_opaque_predicate_python(code: str) -> List[Dict]:

def clean_opaque_predicate_python(code: str) -> List[Dict]:

self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated True -> inlined body"})

self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated False -> removed or replaced with else"})

raise ValueError("unsafe expr")

def detect_opaque_predicate_clike(code: str, lang="C-like") -> List[Dict]:

def clean_opaque_predicate_clike(code: str) -> List[Dict]:

- if condition is always true: replace `if(cond){block}else{else}` -> keep block

-> # Opaque Predicate Complex Test Cases

raise ValueError("Not constant-evaluable")

def detect_opaque_predicate_python(code: str) -> List[Dict]:

def clean_opaque_predicate_python(code: str) -> List[Dict]:

self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated True -> inlined body"})

self.changes.append({"original": orig, "cleaned": cleaned, "reason": "Opaque predicate evaluated False -> removed or replaced with else"})

raise ValueError("unsafe expr")

def detect_opaque_predicate_clike(code: str, lang="C-like") -> List[Dict]:

def clean_opaque_predicate_clike(code: str) -> List[Dict]:

- if condition is always true: replace `if(cond){block}else{else}` -> keep block

===== API Redirection =====

No cases detected.

===== Mixed Language Obfuscation =====

No cases detected.