CFO DOCUMENTATION

Engine.py

Overview

The Engine.py module serves as the entry point for the smali control flow analyzer. It takes a .smali file as input, parses it using ANTLR-generated lexer and parser, builds a CFG, and writes the result to a structured JSON file.

Functions

- ➔ **parse_smali_file(smali_file: str) -> str**
  - ◆ Parses the input .smali file and generates a CFG in JSON format.
  - ➔ Arguments:
    - smali_file: Path to the input smali file
  - ➔ Returns:
    - JSON string representing the CFG
  - ◆ Flow:
    - Loads file as UTF-8 encoded stream
    - Uses SmaliLexer and SmaliParser to generate a parse tree
    - Instantiates CFGBuilder and walks the parse tree to build CFG
    - Creates a CFGVisualizer and returns the serialized JSON CFG
- ➔ **store_to_file(output: dict, smali_file: str) -> None**
  - ◆ Writes the generated CFG JSON to an output file.
    - Arguments:
      - output: Parsed CFG as a dictionary
      - smali_file: Input file name (used to name output)
    - Returns:
      - Creates json_outputs/ directory if missing
      - Writes JSON to json_outputs/<input_filename>.json

CLI Entry Point

1. The script supports command-line usage: `python Engine.py path/to/file.smali`
2. If the file path is not supplied, it prints usage instructions.

Module Dependencies

1. sys, os, json: File and argument handling
2. antlr4: Smali grammar lexer/parser
3. SmaliInfoExtractor: CFG generation logic
4. Trees: For optional debug output of parse tree

SmaliInfoExtractor.py

Overview

The SmaliInfoExtractor.py module is a comprehensive parser listener and control flow graph (CFG) builder for analyzing smali files (the human-readable representation of Android bytecode). It is primarily designed to extract information from the ANTLR parse tree and transform the parsed instructions and directives into an intermediate control flow representation that can later be visualized or exported as JSON.

This module consists of:

- Data structures for blocks and CFG
- A CFGBuilder class that listens to parse events from ANTLR
- Utility classes to format and output the CFG

1. Class: Block

Represents a basic block in the control flow graph.

Attributes:

- **instructions**: List of instruction strings
- **directives**: Smali directives (.method, .end method, etc.)
- **label**: Label assigned to the block (e.g., :**cond_0**)
- **is_entry**: Flag indicating if this block is the start of a method
- **is_exit**: Flag indicating if this block ends a method
- **is_method_start**: Indicates whether this block starts a method
- **register_info**: Dictionary storing local and register count
- **register_types**: Inferred types per register (optional)

2. Class: CFG

Container for all blocks and edges of the control flow graph.

Attributes:

- **blocks**: List of Block objects
- **edges**: Set of tuples (source_index, destination_index) representing edges
- **entry_block**: Index of the entry block
- **in_switch**, **switch_instructions**: Helpers for switch handling

- **in_annotation**, **annotation_buffer**: Buffers and flags for processing annotations

3. Class: CFGBuilder

This is an ANTLR listener that constructs the CFG during parse tree traversal.

Attributes:

- **current_block**: The active block while parsing
- **blocks**: List of all completed blocks
- **method_label_maps**: Maps method names to label-to-block mappings
- **method_start_indices**, **method_end_indices**: Block index bounds for methods
- **block_methods**: Maps block index to method name
- **inferred_types_per_method**: Tracks type information per method/register
- **class_directives**: Captures class-level directives like .class, .super
- **annotation_set**: Used to deduplicate annotations

Listener Event Handlers

Each of these methods corresponds to a grammar rule in the ANTLR Smali parser.

➔ **enterClassDirective** - Extracts class-level information (e.g., **.class**, **.super**, **.source**). Creates a new block if the current one is not empty.
➔ **enterSuperDirective** - Parses and stores the .super directive in **class_directives** and in the current block.
➔ **enterSourceDirective** - Handles **.source** directive. Ensures it goes only into **class_directives**, not the block.
➔ **enterFieldDirective** - Parses **.field** declarations and appends them to **class_directives**.
➔ **enterImplementsDirective** - Handles interfaces (e.g., **.implements**). Appends to **class_directives**.
➔ **enterLocalEndDirective**, **exitLocalDirective** - Handles **.end local** and **.local** lines. Treated as regular instructions in the current block.
➔ **enterLineDirective** - Parses **.line** information for debugging purposes.
➔ **enterCatchDirective**, **enterCatchAllDirective** - Captures exception handler blocks using **.catch** and **.catchall**.
➔ **enterAnnotationDirective**, **exitAnnotationDirective** - Parses annotations and ensures proper structure:
➔ Extracts scope, type, values
➔ Rewrites malformed **EnclosingMethod** annotations
➔ Deduplicates class-level annotations
➔ **enterParamDirective**, **exitParamDirective** - Handles **.param** blocks, collects annotations within a parameter scope, and formats properly.
➔ **exitAnnotationField** - Cleans up formatting for key-value pairs inside annotations.

- ➔ **enterArrayDataDirective**, **exitArrayDataDirective** - Handles **.array-data** blocks:
  - ◆ Extracts numeric values and optional suffixes
  - ◆ Differentiates between in-method and class-level blocks
- ➔ **exitEndFieldDirective** - Handles **.end field**, appends it to class directives.
- ➔ **exitInstruction** - Processes actual bytecode instructions:
- ➔ Splits blocks on control instructions (goto, return, if-*)
- ➔ Preserves formatting for **invoke-*** and **const-***
- ➔ Parses **.registers** and **.locals**
- ➔ **enterLocalsDirective**, **enterRegistersDirective** - Extracts **.locals** and **.registers** info and stores in block metadata.
- ➔ **enterLineLabel** - Handles labels like **:cond_0**. If the current block is not empty, starts a new block. Also updates label mapping for the current method.
- ➔ **enterMethodDirective**, **exitMethodDirective** - Manages the beginning and end of method blocks:
  - ◆ Registers the method name
  - ◆ Tags blocks as entry/exit
  - ◆ Collects all blocks between **.method** and **.end method**
- ➔ **enterPackedSwitchDirective**, **exitPackedSwitchDirective** - Captures packed-switch statements:
  - ◆ Reads packed-switch target table and stores in its own block
  - ◆ Later used for CFG edges
- ➔ **enterSparseSwitchDirective**, **exitSparseSwitchDirective** - Parses **.sparse-switch** values and labels

4. Internal Helper Methods

- ➔ **_build_cfg_edges()** - Connects blocks using control flow edges based on:
  - ➔ goto, if-*, packed-switch, sparse-switch
  - ➔ Adds fall-through edges where necessary
  - ➔ Switch edges are linked based on case labels
- ➔ **_propagate_method_names()** - Ensures every block is associated with a method using entry and exit markers.
- ➔ **_verify_edges()** *(unused currently)* - Asserts whether all conditional and goto instructions have appropriate outgoing edges.

5. Class: CFGVisualizer - Generates JSON representation of the CFG.

- ➔ **generate_json()**
  - ◆ Returns a fully annotated JSON object with:
    - Class-level directives
    - Block list
    - Edges
    - Metadata per block (method, entry/exit, instructions, successors)
- ➔ **_get_successors(block_idx)**
  - ◆ For a given block index, returns all control flow successors with metadata:
    - ➔ Branch type: jump, fall-through, True/False branch
    - ➔ Case values for switch blocks

- ➔ **_format_instructions(instructions)** - Cleans instruction strings, adds spacing and normalizes format for human readability.
- ➔ **_get_branch_type(src, dst)** - Determines the type of edge (True, False, fall-through, jump) based on the last instruction.

---

restructure_cfg.py

Overview

This module is responsible for post-processing the JSON output produced by Engine.py. It groups all control flow blocks under their respective method names, resulting in a more logically organized representation of the smali program.

restructure_cfg_by_method(filepath: str) -> None

Purpose: Reads a JSON CFG, restructures the blocks by method, and writes the output to restructured_jsons/.

Arguments: filepath: Path to the JSON file generated by Engine.py

Behavior:

1. Parses the JSON and indexes blocks by their ID.
2. Identifies method-end blocks (those containing .end method) and clears their successors.
3. Specially handles return instructions by pruning invalid next_block successors.
4. Groups all blocks under their associated method name.
5. Writes the final output to restructured_jsons/<original_name>_restructured.json.

Output Example

Input: JSON from `Engine.py`

```
{
  "class_directives": [
    ".class public Lcom/example/Sample;",
    ".super Ljava/lang/Object;"
  ],
  "blocks": [
    {
      "id": 0,
      "method": "<init>",
      "is_entry": true,
      "is_exit" : false,
      "instructions": ["invoke-direct {p0},
Ljava/lang/Object;-><init>()V"],
```

```
      "directives": [".method public constructor <init>()V"],
      "next_block": [
            {
            "target_block": 1,
            "type": "fall-through"
            }
        ]
    },
    {
      "id": 1,
      "method": "<init>",
      "is_exit": false,
      "instructions": ["return-void"],
      "directives": [".end method"],
      "next_block": [{"target_block": 1,"type": "fall-through"}],
      "register_info": {"locals": "0"}
    },
    {
      "id": 2,
      "method": "getData",
      "is_entry": true,
      "instructions": ["return-object v0"],
      "directives": [".method public getData()Ljava/lang/Object;",
".end method"],
      "next_block": []
    }
  ]
}
```


Output: JSON from `restructure_cfg.py`


```
{
  "class_directives": [
    ".class public Lcom/example/Sample;",
    ".super Ljava/lang/Object;"
  ],
  "<init>": {
    "blocks": [
      {
        "id": 0,
        "method": "<init>",
        "is_entry": true,
        "is_exit": false,
```

```
      "instructions": ["invoke-direct
{p0},Ljava/lang/Object;-><init>()V"],
        "directives": [".method public constructor <init>()V"],
        "next_block": [{"target_block": 1,"type": "fall-through"}]
      },
      {
        "id": 1,
        "method": "<init>",
        "is_exit": false,
        "instructions": [
          "return-void"
        ],
        "directives": [
          ".end method"
        ],
        "next_block": [],
        "register_info": {
          "locals": "0"
        }
      }
    ]
  },
  "getData": {
    "blocks": [
      {
        "id": 2,
        "method": "getData",
        "is_entry": true,
        "instructions": [
          "return-object v0"
        ],
        "directives": [
          ".method public getData()Ljava/lang/Object;",
          ".end method"
        ],
        "next_block": []
      }
    ]
  }
}
```

obfuscator.py

Overview

The obfuscator.py module takes a restructured JSON CFG (produced by restructure_cfg.py) and injects opaque predicates in place of goto instructions to obfuscate control flow. The opaque logic replaces simple jumps with a sequence of arithmetic and logical instructions that obscure the control intent while preserving the execution flow.

This module is central to the control obfuscation toolchain.

Functions:

➔ **transform_gotos_with_opaque_blocks_grouped(json_data, stats):**

- ◆ Purpose: Iterate through each method in the grouped CFG and apply transformation logic to replace goto instructions with obfuscated instruction blocks.
- ◆ Returns:
  - A new JSON object where the control flow has been rewritten to include opaque logic.

➔ **transform_method_blocks(...):**

- ◆ For each block in a method:
- ➔ It checks if the method contains any goto instructions.
- ➔ If present, the goto is replaced by a complex opaque conditional logic chain:
  - ◆ Uses a safe or unused register.
  - ◆ Calculates dummy arithmetic values.
  - ◆ Inserts misleading branches (if-lez, etc.) with fake labels.
  - ◆ Performs dead-end method calls and more confusing patterns.
- ➔ Updates the block's instruction list and removes next_block.

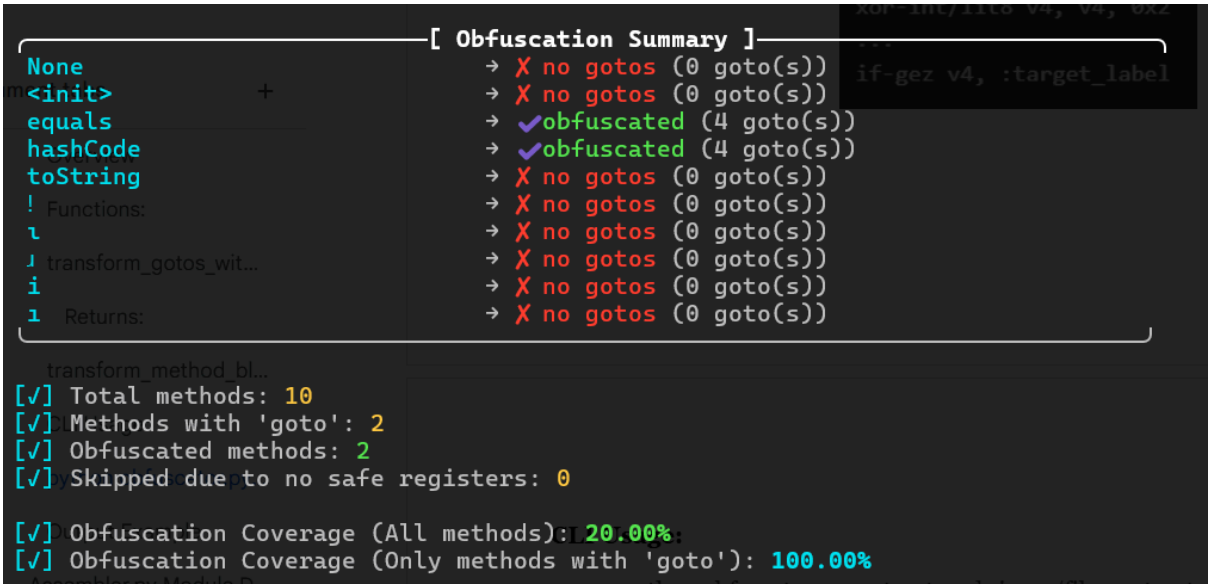
```
const/16 v4, 0x1F
shl-int/lit8 v4, v4, 2
add-int/lit8 v4, v4, 7
xor-int/lit8 v4, v4, 0x2
...
if-gez v4, :target_label
```

CLI Usage:

python obfuscator.py restructured_jsons/file_restructured.json

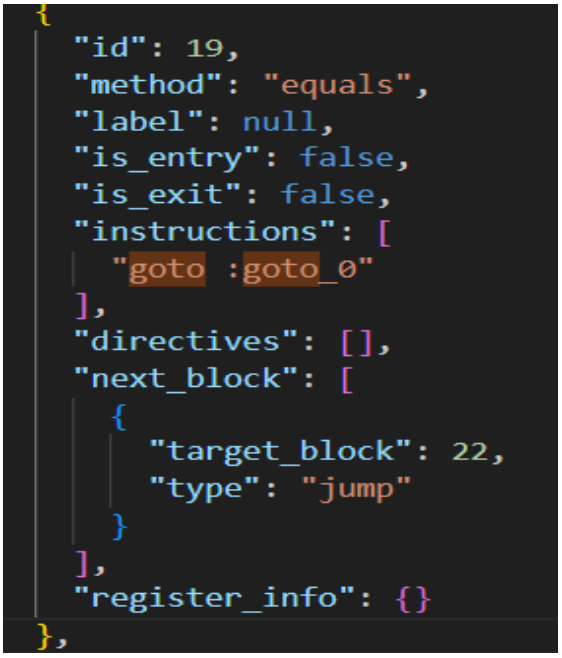
This will generate:

And print a summary:



Output Example

Given a restructured JSON with:



The output will replace the goto with a sequence like:


```
{
  "id": 19,
  "method": "equals",
  "label": null,
  "is_entry": false,
  "is_exit": false,
  "instructions": [
    "const/16 v4, 0x1F",
    "shl-int/lit8 v4, v4, 2",
    "add-int/lit8 v4, v4, 7",
    "xor-int/lit8 v4, v4, 0x2",
    "mul-int/lit8 v4, v4, 5",
    "shl-int/lit8 v4, v4, 1",
    "add-int/lit16 v4, v4, 11981",
    "shl-int/lit8 v4, v4, 2",
    "add-int/lit16 v4, v4, 1267",
    "xor-int/lit16 v4, v4, 0x1234",
    "if-gez v4, :goto_0",
    "const/16 v4, 0x3A",
    "shl-int/lit8 v4, v4, 3",
    "xor-int/lit16 v4, v4, 0x55AA",
    "add-int/lit16 v4, v4, 1234",
    "rem-int/lit8 v4, v4, 7",
    "mul-int/lit8 v4, v4, 19",
    "or-int/lit16 v4, v4, 0x3333",
    "and-int/lit16 v4, v4, 0x7FFF",
    "if-lez v4, :fake_false_0",
    "goto :fake_continue_0",
    ":fake_false_0",
    "const/4 v4, 0x0",
    "goto :fake_continue_0",
    ":fake_continue_0",
    "invoke-static {v4}, Ljava/lang/Integer;->toHexString(I)Ljava/lang/String;",
    "add-int/lit8 v4, v4, 3",
    "xor-int/lit8 v4, v4, 0x3",
    "shl-int/lit8 v4, v4, 1",
    "goto :goto_0"
  ],
  "directives": [],
  "register_info": {}
},
```

Assembler.py

Overview

The Assembler.py script reads the transformed (or original) CFG JSON and reconstructs it into .smali files. It supports two modes:

- Normal: Directly uses JSON from Engine.py
- Injected: Used when JSON is grouped by method (e.g., output from restructure_cfg.py or obfuscator.py)

Function:

➔ **flatten_restructured_json(cfg)**
- ◆ If the CFG JSON is grouped by methods, it flattens it back to a single blocks list (the format expected by the assembler).

➔ **Assembler(cfg, output_file):**
- ◆ Reconstructs the .smali file from the flattened CFG JSON.
- ◆ Writes:
  - Class-level directives
  - Each method's start directives (.method, .locals, .registers, .param)
  - All instructions and labels
  - .end method block for method termination

CLI Usage:

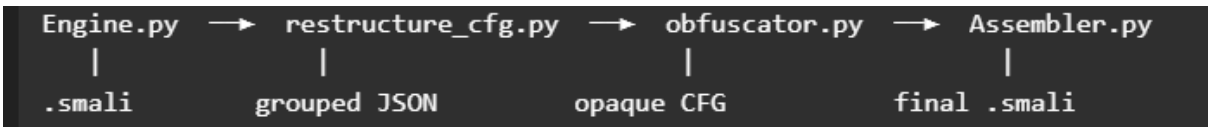
python Assembler.py obfuscated_jsons/obfuscated_file.json --injected

Output:

assembled_smali/modified_obfuscated_file.smali

Note : Without --injected, it assumes the JSON is in flat Engine.py style.

Summary Flow:



packer.py

Overview

The packer.py script is the final step in the control obfuscation toolchain. It takes a directory containing modified smali code (typically reconstructed from obfuscated JSON using Assembler.py), recompiles the APK using apktool, and signs it using the uber-apk-signer tool. This allows the modified APK to be installed and run on Android devices.

This module automates both recompilation and signing.

Function:

- ➔ **recompile_apk(temp_dir, output_apk_name="modified_output.apk")**
  - ◆ Purpose: Rebuilds a decompiled APK directory using apktool.
  - ◆ Arguments:
    - temp_dir: Path to the directory containing modified smali code and resources.
    - output_apk_name: Output APK file name (default is modified_output.apk).
  - ◆ Behavior:
    - Calls apktool b to recompile the smali code and resources.
    - Stores the final APK in the current or specified path.
    - Assumes apktool.jar is located at ./tools/apktool.jar.

- ➔ **apk_signing(apk_path)**
  - ◆ Purpose: Signs the newly compiled APK using uber-apk-signer.
  - ◆ Arguments:
    - apk_path: Path to the unsigned APK.
  - ◆ Behavior:
    - Invokes uber-apk-signer-1.3.0.jar with --overwrite.
    - Looks for the signed APK in the output/ directory.
    - Moves the signed APK back to the original path, replacing the unsigned one.
    - Deletes the output/ directory to clean up.

CLI Usage

python packer.py <path_to_decompiled_apk_folder>

Where **<path_to_decompiled_apk_folder>** contains the smali code and resources to be packed.

Result:

- Recompiled APK at: ./modified/<apk_name>.apk
- Signed and ready to install

---

run.py

Overview

The run.py script is the central orchestrator of the entire control flow obfuscation pipeline. It takes two inputs: the target APK file path and the package name to narrow down smali files. It automates every step in the pipeline:

- Decompiling the APK using apktool
- Filtering smali files using package name and relevant keywords
- Generating control flow graph (CFG) using Engine.py
- Structuring the CFG using restructure_cfg.py
- Obfuscating control flow using obfuscator.py
- Reassembling obfuscated smali using Assembler.py
- Repacking and signing the modified APK using packer.py
- Summarizing obfuscation stats and APK size change

Features

- Filters smali files inside the specified package directory based on keywords like **activity, fragment, view, camera**
- Tracks goto statements and replaces them with opaque control flow
- Displays a color-coded summary of:
    - Total smali files scanned and matched
    - Number of files/methods obfuscated
    - Coverage percentage (by smali and by method)
    - APK size comparison before and after

Main Execution Steps

Step 1: Setup and Decompilation

- Validates input arguments
- Creates a temporary working directory
- Decompiles APK using apktool into temp directory

Step 2: Smali Filtering

- Scans the decompiled smali/ directory
- Filters smali files based on package name and keywords
- Displays matched files

Step 3: Per-File Pipeline Execution

For each matched smali file:

1. CFG Generation: Calls Engine.py to generate control flow JSON
2. CFG Structuring: Uses restructure_cfg.py to group blocks by method
3. Obfuscation: Runs obfuscator.py with opaque predicates
    - Gathers method-level stats (e.g., total gotos, number obfuscated)
4. Assembly: Calls Assembler.py with --injected to reconstruct .smali

5. Copy Back: Replaces the original smali in the temp directory

Step 4: Summary Statistics

- Calculates:
  - Total matched smali
  - Methods with goto
  - Methods obfuscated
  - Smali files obfuscated
  - Obfuscation coverage (by method and smali)
- Prints a color-coded summary table

Step 5: APK Repacking and Signing

- Calls packer.py with the modified temp directory
- Signs APK using uber-apk-signer
- Compares original and modified APK sizes

CLI Usage

python run.py <path_to_apk> <java_package_name>

Summary Flow (Full Pipeline)

```
run.py
  ├── Engine.py           → Generate flat CFG
  ├── restructure_cfg.py  → Group blocks by method
  ├── obfuscator.py       → Inject opaque control flow
  ├── Assembler.py        → Build obfuscated .smali
  └── packer.py           → Recompile & sign APK
```

Output Example

```
🔍 Matched smali files: 3
  -> smali/com/example/myapp/MyActivity.smali
  -> smali/com/example/myapp/CameraView.smali

============================== CONTROL FLOW OBFUSCATION SUMMARY ======
🔹 Total smali matched:                 3
🔹 Smali files containing 'goto':       2
🔹 Smali files obfuscated:              ✓ 2
🔹 Total methods obfuscated:            4
🔹 Total methods with 'goto':           5
✅ Obfuscation coverage (smali):        66.67% [2/3]
✅ Coverage (methods with goto):        80.00% [4/5]
✅ Smali obfuscation (where goto exists): 100.00% [2/2]
================================================================================

Original APK size: 3.82 MB
Modified APK size: 4.04 MB
📈 Size increased by: 0.22 MB
```