Big picture: what EtherSolve is doing

Input: EVM bytecode as a hex string
Output: `Cfg` object containing:

- a `TreeMap<Long, BasicBlock>` keyed by byte offset
- edges via `BasicBlock.successors`
- metadata: `CfgBuildReport` + `remainingData` (pruned metadata / trailing bytes)
- an explicit **EXIT** block ("super node") that all terminal blocks connect to

The build process has **two stages of edge discovery**:

1. **Easy edges**: resolve `JUMP/JUMPI` targets when the target is a static `PUSH` right before the jump.
2. **Hard edges**: resolve "orphan jumps" via **symbolic execution stack** to infer runtime jump destination.

---

1) `BasicBlock.java`

A `BasicBlock` is a chunk of bytecode (inherits from `parseTree.Bytecode`) plus CFG wiring.

Key fields

- `ArrayList<BasicBlock> predecessors`
- `ArrayList<BasicBlock> successors`
- `int stackBalance`
- `BasicBlockType type` (COMMON/DISPATCHER/FALLBACK/EXIT)

Stack balance

The block maintains `stackBalance` (net stack delta if the block executes fully).

- When adding one opcode: updates incrementally:
  - `stackBalance -= consumed`
  - `stackBalance += generated`
- When adding multiple: recomputes using `calculateStackBalance()`

This is useful later for stack sanity or analyses (though in the provided builder it's not used for edge logic directly).

CFG wiring

`addSuccessor(next)` updates both sides:

- `this.successors.add(next)`
- `next.predecessors.add(this)`

`toString()`

Special-cases exit blocks: `"<offset>: EXIT BLOCK"`.

---

2) `BasicBlockType.java`

Just a label enum:

- `COMMON`: default
- `DISPATCHER`: blocks before the “end of dispatcher region” heuristic
- `FALLBACK`: fallback detection heuristic
- `EXIT`: artificial super node

---

3) `Cfg.java`

Wrapper over the graph + build metadata.

Main members

- `TreeMap<Long, BasicBlock> basicBlocks` (sorted by offset)
- `Bytecode bytecode` (original parsed bytecode)
- `CfgBuildReport buildReport`
- `String remainingData` (stuff not modeled as code)

Useful helper

`getSuccessorsMap()` returns adjacency list:
`offset -> [successor offsets...]`

Also implements `Iterable<BasicBlock>` over block values.

---

4) `CfgBuildReport.java`

A structured error/metrics recorder for CFG building.

It tracks error categories that correspond directly to the builder stages:

Jump resolution errors

- `directJumpTargetErrors`: static push target wasn't a block
- `orphanJumpTargetNullErrors`: symbolic jump points to non-existent block
- `orphanJumpTargetUnknownErrors`: symbolic stack top unknown at jump
- `loopDepthExceededErrors`: DFS too deep (LOOP_DEPTH bound)
- `blockLimitErrors`: DFS explored too many blocks (BLOCK_LIMIT bound)

CFG sanity errors

- `multipleRootNodesErrors`: validator found more than one entry/root
- `stackExceededErrors`: symbolic execution stack > 1024
- `criticalErrors`: caught exceptions during build

It also keeps a full `errorLog` string.

---

5) `CfgBuilder.java` — the whole pipeline

This is the heart. The public entrypoint is:

```
public static Cfg buildCfg(String binary)
```

5.1 Delimiters and heuristics

Basic block delimiters

A block ends when encountering any of:

- `JUMP`, `JUMPI`
- `STOP`, `REVERT`, `RETURN`
- `INVALID`, `SELFDESTRUCT`

Plus, a `JUMPDEST` can start a new block (if current block is non-empty).

Global safety bounds

- `LOOP_DEPTH = 1000` : symbolic DFS depth limit
- `BLOCK_LIMIT = 200000` : max explored blocks in symbolic stage

- `REMOVE_ORPHAN_BLOCKS = true` : optional post-prune

5.2 `buildCfg(binary)` flow (step-by-step)

Step A — preprocess bytecode (library + child contracts stripping)

- If bytecode matches `^73{40}3014...$` pattern, it strips "library prefix" logic.
- Splits child contracts via regex: `(?=(60(60|80)604052))`
  - treats second segment as "children contracts"
- Parses only the primary segment as bytecode

Step B — parse bytecode

```
Pair<Bytecode, String> sourceParsed = BytecodeParser.parse(...)
Bytecode bytecode = sourceParsed.getKey();
String remainingData = sourceParsed.getValue() + childrenContracts;
```

So parser returns:

- decoded opcode stream in `Bytecode`
- leftover bytes ("metadata" or non-code tail)

Step C — build blocks: `generateBasicBlocks(bytecode)`

This does a linear scan over opcodes and splits blocks:

Rules:

1. If opcode is a **delimiter**:
   - include it in current block
   - store block
   - start new block at `o.getOffset() + 1`
2. Else if opcode is `JUMPDEST` and current block isn't empty:
   - store current
   - start new block at `JUMPDEST` offset
   - include the `JUMPDEST` in new block
3. Else:
   - keep adding opcode to current block

Result is a `TreeMap<offset, block>`.

Step D — add easy successors: `calculateSuccessors(...)`

For each block, it inspects the **last opcode**:

Case 1: `JUMP` with `PUSH` immediately before it

Resolve direct jump:

- destination = pushed constant
- add edge to that block if exists, else record `directJumpTargetErrors`

Case 2: `JUMPI`

Adds **two successors** if possible:

- fallthrough edge: next block at `lastOffset + lastOpcodeLength`
- conditional jump edge if `PUSH` before it resolves to a known block

If target doesn't exist: `directJumpTargetErrors`.

Case 3: other delimiters

No successor (terminal).

Case 4: normal block

Fallthrough to next block by offset.

So after this stage:

- some jumps remain "orphan" (dynamic jump target not known)

Step E — resolve orphan jumps via symbolic execution: `resolveOrphanJumps(...)`

This is the advanced phase.

It performs a DFS-like traversal where the state is:

- current `BasicBlock`
- a **SymbolicExecutionStack** (copied on branching)
- current depth

It executes all opcodes in the block to track stack state, then:

If last opcode is a `JumpOpcode` (dynamic `JUMP`/`JUMPI` style)

It tries:

```
nextOffset = stack.peek().longValue()
```

- If unknown → `orphanJumpTargetUnknownErrors`
- If offset doesn't map to a block → `orphanJumpTargetNullErrors`
- If valid → add successor edge

Then it executes the last opcode too (to keep stack consistent).

DFS expansion logic

- It uses a `visited` set of `(srcOffset, dstOffset, stackState)` (Triplet)
  - intent: avoid revisiting same edge under same symbolic context
- Branching:
  - if not a JumpOpcode: enqueue all successors
  - if JumpOpcode and nextOffset resolved: enqueue that single successor

Guards:

- if depth reaches `LOOP_DEPTH`: log `loopDepthExceededErrors`
- if explored blocks hits `BLOCK_LIMIT`: log and stop

Net effect: this phase recovers edges for computed jumps, e.g., dispatcher jump tables.

Step F — remove remaining data: `removeRemainingData(...)`

It looks for blocks that:

- have **no predecessors**
- and whose bytes are `"fe"` (INVALID opcode marker)
  When found, it treats that offset as start of "non-code / data".

Then it removes all blocks with offset ≥ `firstInvalidBlock`,
and returns the sliced byte tail from that offset.

Step G — remove "orphan blocks" (optional): `removeOrphanBlocks(...)`

Only runs if `buildReport.getTotalJumpError() == 0`.

It does a reachability DFS from the entry block, finds the **max offset reachable**, and deletes all blocks with offset > that.

Updates remaining data accordingly.

Step H — detect dispatcher: `detectDispatcher(...)`

Heuristic:

- find highest block offset that ends with `RETURN` or `STOP`

- mark **every block with offset <= that offset** as `DISPATCHER`

This is a coarse "everything before runtime code split" style heuristic.

Step I — detect fallback: `detectFallBack(...)`

Heuristic:

- take entry block's highest-offset successor (max successor)
- then that block's highest-offset successor (max second successor)
- pick that as "fallback candidate"
- if candidate is only a `JUMPDEST` block, mark its successors instead
- do not mark anything that ends with `REVERT`

Step J — validate: `validateCfg(...)`

Counts roots = blocks with zero predecessors.
If roots != 1 → `multipleRootNodesErrors`

Step K — add super node: `addSuperNode(...)`

Creates a final synthetic block:

- offset = lastKey + lastBlockLength
- type = `EXIT`
  Then for every block with no successors, add successor → EXIT
  Finally, insert EXIT block into map.

This guarantees a single sink for graph algorithms.

How EtherSolve differs from the Python CFG builders

EtherSolve CFG nodes:

- **basic blocks of EVM opcodes** (byte offsets)
- edges computed from EVM control flow semantics (JUMP/JUMPI/STOP/RETURN/etc.)
- dynamic edges resolved via **symbolic stack execution**

python-graphs / staticfg:

- nodes are Python AST-level constructs or statement blocks
- edges derived from Python semantics, not runtime bytecode
- no “jump tables” style dynamic resolution needed (except exceptions etc.)

Practical “gotchas” in EtherSolve’s approach

1. **Static jump resolution is limited**
   Only resolves `JUMP/JUMPI` targets when the opcode right before is a `PUSH`.
   Many real contracts compute jump destinations through arithmetic, DUP/SWAP, etc — hence the need for symbolic stage.
2. **Symbolic stage can explode**
   Hence `LOOP_DEPTH` and `BLOCK_LIMIT`. On large contracts, you can end up with incomplete CFG.
3. **`visited` includes stack state**
   That’s good for correctness, but means fewer merges → larger exploration space.
4. **Fallback/dispatcher detection is heuristic**
   These labels are helpful for analysis, but they’re not guaranteed correct across compiler versions / patterns.