VEIL: Reading Control Flow Graphs Like Code

PHILIPP SCHAAD, ETH Zurich, Switzerland
TAL BEN-NUN, Lawrence Livermore National Laboratory (LLNL), USA
TORSTEN HOEFLER, ETH Zurich, Switzerland

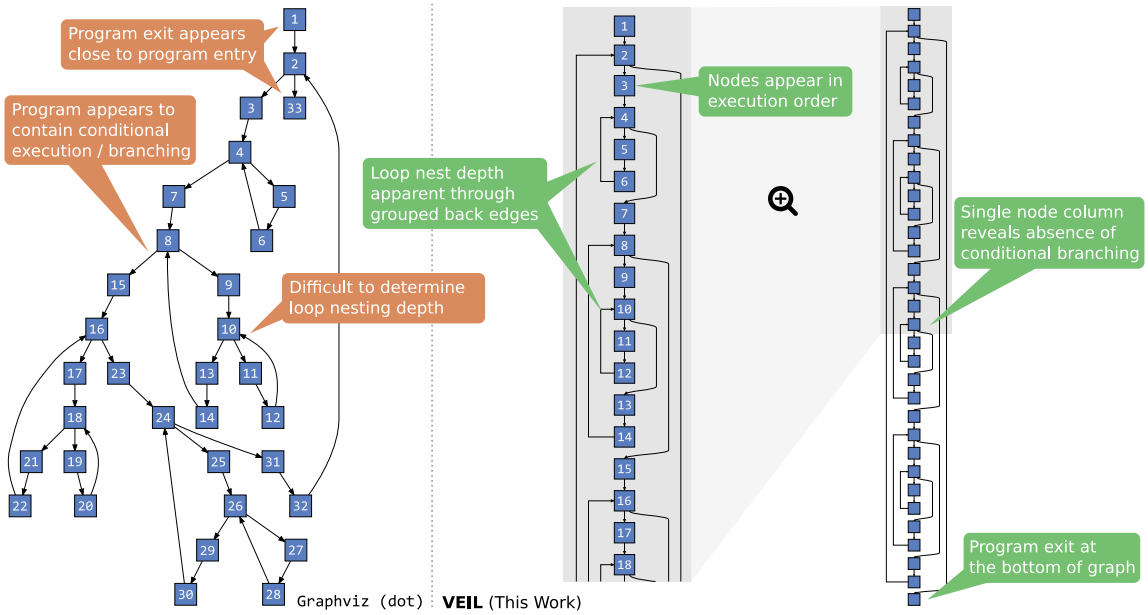


Fig. 1. The control flow graph of a program computing 2-D electromagnetic wave propagation, drawn using Graphviz with the dot layout algorithm (left) and our new layout algorithm, VEIL (right), which optimizes layouts for control flow graphs.

Control flow graphs (CFGs) are essential tools for understanding program behavior, yet the size of real-world CFGs makes them difficult to interpret. With thousands of nodes and edges, sophisticated graph drawing algorithms are required to present them on screens in ways that make them readable and understandable. However, being designed for general graphs, these algorithms frequently break the natural flow of execution, placing later instructions before earlier ones and obscuring critical program structures. In this paper, we introduce a set of criteria specifically tailored for CFG visualization, focusing on preserving execution order and making complex structures easier to follow. Building on these criteria, we present VEIL, a new layout algorithm that uses dominator analysis to produce clearer, more intuitive CFG layouts. Through a study of CFGs from real-world applications, we show how our method improves readability and provides improved layout performance compared to state of the art graph drawing techniques.

CCS Concepts: • **Human-centered computing** → **Graph drawings**; *Information visualization.*

Additional Key Words and Phrases: Control Flow Graphs, Layout Algorithm, Readability and Comprehension

Authors' Contact Information: Philipp Schaad, philipp.schaad@inf.ethz.ch, ETH Zurich, Zurich, Switzerland; Tal Ben-Nun, talbn@llnl.gov, Lawrence Livermore National Laboratory (LLNL), Livermore, USA; Torsten Hoefler, htor@inf.ethz.ch, ETH Zurich, Zurich, Switzerland.

1 Introduction

Control flow graphs (CFGs) are directed graphs that represent the order of execution in a program. Visual analysis and understanding of CFGs are central to many tasks in computer systems, including debugging [18, 21], performance tuning [13, 29, 46], compiler development [17, 69], reverse engineering [43, 56], and security analysis [1, 2], to name a few. In many of these contexts, CFGs are not just a convenient but necessary tool. For instance, when source code is unavailable during binary analysis, or when compiler optimizations alter original control structures such as through simplification or reordering. However, the CFGs of real-world applications are often very large, containing thousands of nodes and edges. This scale makes visual comprehension difficult, especially in the presence of complex control flow constructs such as loops with multiple exit paths, extensive branching, or `goto` statements.

To support understanding and analysis, numerous *graph drawing* algorithms have been developed to map graphs to the 2-D plane. Considerable research has focused on defining graph layout criteria to maximize readability, often validated through comprehensive user studies [8, 52, 53, 60, 68]. This has led to the development of specialized graph drawing algorithms for specific classes of graphs, as well as some general-purpose algorithms that have become widely adopted in tools such as Graphviz [27, 32, 33].

While general-purpose algorithms perform well across many domains, they often struggle with complex CFGs. Their adherence to broad layout criteria can introduce artifacts that reduce readability. For example, nodes that appear later in the execution order may be placed closer to the start of the program than nodes appearing earlier in execution order, disrupting the natural flow of analysis (see Fig. 1). Such artifacts suggest that state-of-the-art layout criteria are insufficient for the domain of CFGs and program flow analysis.

In this paper, we codify the criteria a graph drawing algorithm geared towards control flow graphs should follow. We identify interaction and visualization techniques that benefit from the outlined criteria. These criteria and techniques aim to help humans interact with CFGs in a more natural manner, more comparable to interactions with program source code rather than typical graph representations.

We develop a novel graph drawing algorithm called VEIL, the Vertical Execution-order Informed Layout, that optimizes CFG drawings with respect to the identified criteria. By employing graph dominator analysis, our algorithm not only achieves intuitive graph layouts, but offers improved layout times compared to graph drawing algorithms frequently used for CFGs.

Using a series of CFGs from real-world applications, we perform an in-depth evaluation of our graph drawing algorithm and compare it to state-of-the-art techniques. We evaluate and compare how well different techniques adhere to the identified layout criteria, and discuss how this impacts readability and comprehension.

In summary, this paper makes the following contributions:

- A codified set of aesthetics criteria for evaluating CFG-specific graph layouts.
- VEIL, a novel layout algorithm that optimizes for CFG-specific aesthetics and structural criteria.
- An in-depth comparison of VEIL with state-of-the-art algorithms across real-world CFGs.

2 Aesthetics of Control Flow Graph Drawings

Over the years, numerous studies have proposed concrete aesthetic criteria that graph drawing algorithms should follow to maximize readability and comprehension [6, 52, 53, 61, 62]. Many of these criteria have been repeatedly validated in empirical user studies [9, 37, 39, 53, 54, 66] and have become de facto standards for graph visualization. By extension, algorithms for drawing control flow graphs (CFGs) should also adhere to these principles. The nine most consistently

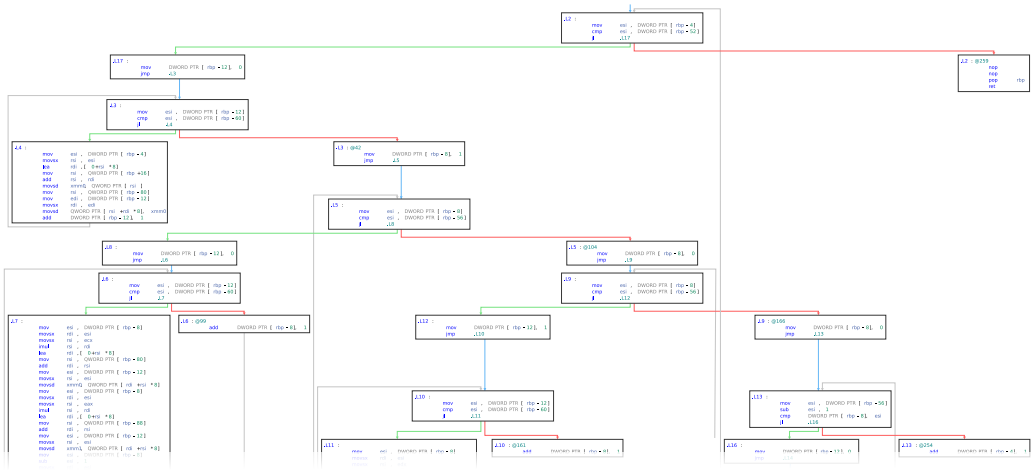


Fig. 2. Excerpt of a CFG as visualized by Compiler Explorer [11], showing x86 assembly instructions in each basic block.

supported criteria, i.e., those that appear frequently across studies and are rated as important for comprehension, are as follows:

C1 (Node Orthogonality)**.** Nodes should be evenly distributed on a virtual Cartesian grid without overlap, except when representing hierarchical nesting [20, 38, 52, 53, 68].

C2 (Edge Orthogonality)**.** Edges should run in straight horizontal or vertical lines [8, 52, 53, 68].

C3 (Edge Crossings)**.** The number of edge crossings should be minimized [37, 39, 52–54, 61].

C4 (Edge Bends)**.** The number of edge bends, i.e., changes in direction along an edge, should be minimized [53–55, 61, 66].

C5 (Edge Uniformity)**.** Edge lengths should be kept uniform [15, 40, 41, 62].

C6 (Short Edges)**.** The total edge length, as well as the maximum edge length, should be minimized [61, 66]

C7 (Graph Area)**.** The overall area occupied by the graph should be minimized [53, 61, 62, 68]

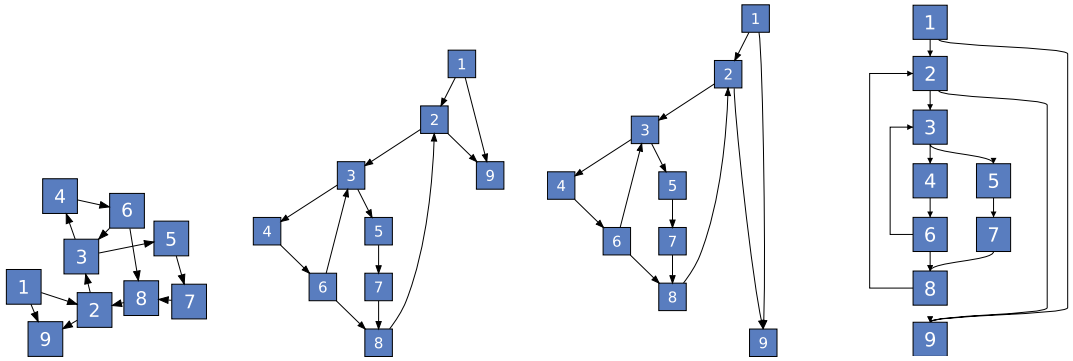
C8 (Symmetry)**.** Local and global symmetry should be maximized [16, 67]

C9 (Consistent Flow)**.** In directed graphs, edges should follow a uniform flow direction [52, 68].

These criteria are not always mutually compatible, and tradeoffs are often necessary depending on the graph class. Consequently, graph drawing algorithms typically optimize for a subset of criteria or adapt them to application-specific needs. For example, prior work has investigated domain-specific aesthetics for object-relation diagrams (e.g., UML [50]) [26, 53, 55, 68] and for network graphs [35, 47, 65]. By contrast, relatively little research has focused on improving layout quality specifically for control flow graphs beyond the general-purpose criteria above.

2.1 Control Flow Graphs and Aesthetics Criteria

CFGs are a special class of directed graphs that capture the order in which operations in a program are executed. A directed graph (digraph) is a pair $G = (V, E)$, where V is a set of nodes and E is a set of ordered pairs of nodes $e = (u, w)$, representing edges directed from u to w.



(a) Force-directed placement, minimizing magnetic forces. (b) Layered graph drawing, optimizing for criteria C1-C9. (c) CFGExplorer [18], enforcing happens-before (C9) for loops. (d) VEIL, enforcing edge direction grouping and all criteria C1-C11.

Fig. 3. CFG of a conditionally executed 2-level loop nest drawn using different layouts to illustrate the effects of aesthetics criteria.

CFGs form a subclass of digraphs with two additional requirements:

(1) A CFG has exactly one source node (a node with no incoming edges), representing the program entry.
(2) A CFG has at least one sink node (a node with no outgoing edges), representing program exit points.

Each node in a CFG corresponds to a *basic block*, which is a sequence of instructions with no branching into or out of the basic block. Edges then represent possible control flow between blocks, encoding essential information about a program's structure and behavior. For example, a node with multiple outgoing edges represents conditional branching (e.g., an `if-else` statement). An example of a CFG can be seen in Fig. 2, which shows the instructions contained in each basic block and contains a number of edges forming loops and branching. For simplicity, most of the CFGs shown in this paper will show basic blocks as abstract nodes with no contents (e.g., Fig. 1, Fig. 3), since the focus lies on the layout itself rather than visualization of instructions.

CFGs also represent a special case of digraphs in that they exhibit a few identifying characteristics. Apart from programs with many large `switch-case` constructs, CFGs typically have a low average degree (few edges per node) and are thus sparse ($|E| \ll |V|(|V|-1)$). In addition to branching, CFGs usually contain *back edges*, i.e., edges that form cycles [36], which typically correspond to loop constructs such as `for` or `while` loops. Both branching and back edges are critical for understanding program behavior, and detecting them reliably is essential for program analysis [17, 56].

To illustrate how crucial established aesthetics criteria are, even when not adapted to CFGs, Fig. 3a shows the CFG of a conditionally executed 2-level loop nest, as laid out by force-directed placement using Graphviz's [27] `fdp` algorithm. Force-directed placement draws graphs by minimizing attractive and repulsive forces between nodes based on their distances to each other and whether they are connected through edges. While this placement keeps edges short, consistent, and straight (C6, C5, C4), and minimizes overall graph area (C7), it does not lead to edges pointing in a consistent direction (C9) or that avoid crossings (C3). By violating both these criteria, readability of the drawing suffers compared to a drawing of the same CFG using a layered graph drawing algorithm (`dot` [32]), shown in Fig. 3b.

2.2 Formalizing Control Flow-Specific Aesthetic Criteria

Using the typical characteristics of CFGs, we formalize two additional control flow-specific aesthetic criteria for drawing CFGs to further improve readability, while still adhering to established graph drawing criteria. When reasoning about execution order, developers are most familiar with source code, which presents instructions as a linear sequence within

a text file. Within a scope (e.g., a function or loop), the perceived execution order typically aligns with textual order, unless altered by compiler or runtime optimizations. Any deviation, such as loops or branching, is explicitly marked with braces, indentations, or keywords, which provide visual cues for scope and control flow.

Given this strong mental model of *top-to-bottom execution order* in source code, laying out a CFG in a similar manner should transitively improve readability and comprehension. We formalize this intuition with the following control flow-specific aesthetic criterion:

C10 (Happens-Before)**.** For any two basic blocks A and B, if B is guaranteed to execute after A (when both are executed), then B should be drawn below A. Conversely, if A must execute before B, then A should appear above B.

This criterion ensures that any block executed after a loop is drawn below the loop body. However, it may lead to longer edges between the loop header (the condition check) and the block immediately following the loop (the loop exit), which conflicts with edge uniformity (C5) and short edges (C6). As a result, most existing algorithms do not enforce C10, as demonstrated in Fig. 3b.

Despite this tradeoff, edge length can provide valuable structural cues *if* C10 is respected. For example:

- A long downward edge indicates a conditional branch that skips a large portion of the program, or, if it reaches the bottom, an early termination, with edge length proportional to the skipped portion of the program.
- A long upward edge (back edge) signals a loop, with its length correlating to loop size.

Fig. 3c shows a drawing of the same CFG as Fig. 3b but using CFGExplorer [18], which attempts to highlight loops, thus better adhering to C10. In doing so, one such long downward edge that was less detectable in Fig. 3b becomes more exposed, indicating a conditional branch skipping most of the program.

When edges are drawn orthogonally with minimal bends (C2, C4), the negative impact of longer edges has further been shown through user studies to be significantly reduced [8, 9]. Thus, longer edges resulting from enforcing happens-before vertical ordering can enhance rather than hinder comprehension.

To strengthen this effect, we propose grouping semantically similar long edges. Assuming C10 is satisfied, three categories of long edges arise naturally:

(1) **Forward edges** that skip parts of the program in the form of conditional branches.
(2) **Back edges** that run upward, forming loops.
(3) **Loop-exit edges** connecting a loop header to the block immediately following the loop (loop exit).

Routing all back edges on one side of the graph makes loops and their approximate size immediately visible. Similarly, routing forward edges to the opposite side, grouped together, highlights conditionally executed code and early program termination. Both classes of edges can be efficiently identified with a depth-first search in the graph [36]. We capture this in a second control flow-specific aesthetic criterion:

C11 (Edge Direction Grouping)**.** Back edges (against program flow) should be grouped on one side of the graph, while forward edges should be grouped on the opposite side.

This highlights any violation of C5 and C9, which carries semantic information in the context of CFGs. A layout that respects both C10 and C11 is expected to be narrow and vertical when control flow is simple, but widen in regions with many conditional branches (e.g., large case distinctions). This property makes structural complexity visually salient and facilitates rapid program understanding at a glance, while allowing graphs to be explored through vertical scrolling similar to text files, rather than panning. Combining this with navigation features such as jumping to a given edge's

start or end point allows for more efficient navigation of large CFGs. An example of a layout respecting C10 and C11 can be seen in Fig. 3d, where the CFG from Fig. 3c is drawn using our new algorithm, VEIL. By grouping back edges to the left, this layout more readily exposes the presence of a 2-level nested loop, and the drawing with C10 indicates conditional branching through a deviation from a single-column layout towards the center.

3 VEIL: Vertical Execution-Order Informed Layout

The VEIL graph drawing algorithm falls into the category of layered graph drawing methods, also known as the Sugiyama Framework [60]. Layered graph drawing arranges nodes of a directed graph into top-to-bottom layers (or ranks), ensuring that all edges, except back edges, point downward, thereby satisfying aesthetic criterion C9 for consistent flow. This well-studied framework forms the basis for graph drawing algorithms in many state-of-the-art tools, most notably Graphviz [27] with the `dot` algorithm [32, 33].

On a high level, layered graph drawing proceeds in three main phases:

(1) **Layer assignment**. Each node is assigned to a rank so that edges point from higher to lower layers.
(2) **Crossing minimization**. Nodes within each layer are reordered to reduce the number of edge crossings. Since finding an optimal layout with the fewest crossings is NP-complete [34], a variety of heuristic methods have been developed to find good approximations [30, 33, 48]. To allow for the use of such methods, edges that span multiple ranks are sub-divided into chains of single-rank edges through the help of virtual intermediate nodes.
(3) **Coordinate assignment**. Based on the layer assignments and node orderings, screen coordinates are assigned to all nodes and edges. Previously split up long edges spanning multiple layers are re-formed, removing virtual intermediate nodes (also referred to as *dummy nodes*) and inserting edge bends in their stead.

This procedure is particularly well suited to directed or hierarchical graph structures, such as CFGs, because it produces layouts that more effectively satisfy widely accepted graph aesthetics criteria (C1-C9; see Section 2) than alternative approaches. Consequently, graph drawing algorithms following the Sugiyama Framework have become the de facto standard for visualizing CFGs [17, 18], including for the widely used compiler toolchain Clang/LLVM [44].

However, the technique is inherently designed for directed acyclic graphs (DAGs), which is why the layer assignment phase typically begins by breaking cycles through back-edge reversal, only re-forming cycles during coordinate assignment. Since CFGs often contain many cycles, this frequently produces layouts that violate the happens-before and edge grouping aesthetics criteria (C10/C11). For example, in a cycle representing a `for`-loop, the control flow after the loop body is often placed on a higher rank than most of the loop, in an attempt to preserve short and uniform edge lengths (C5/C6; see Section 2). Because cycles are removed prior to ranking, the algorithm cannot recover the correct happens-before relationships once cycles are reintroduced. Similarly, during crossing minimization, upward edges cannot be distinguished and therefore cannot be grouped together, making them less identifiable in the final layout.

To improve layouts for CFGs, VEIL leverages the three-phase structure of layered graph drawing, following the principle of orienting edges from higher to lower ranks, but introduces novel algorithms for the individual phases. Specifically, both the layer assignment and crossing minimization phases are modified to account for back edges by leaving them intact. This allows layer assignment to optimize rankings with respect to the happens-before criterion (C10) and enables in-layer permutations to group both back and forward edges (C11). In addition, the coordinate assignment phase is refined to improve node and edge orthogonality (C1/C2), minimize overall graph area (C7), and enhance symmetry (C8). The following subsections describe each phase in turn.

Algorithm 1: VEIL layer assignment algorithm

Data: CFG G with all node ranks initialized to 0, empty queue Q
Result: CFG G with all nodes assigned to their rank

if *number of sink nodes in G > 1* **then**
  Connect all sink nodes to a virtual sink node
end
$Q.push([G.start, 0])$;
while $Q.length > 0$ **do**
  $[v, rank] \leftarrow Q.pop()$;
  $v.rank \leftarrow \max(rank, v.rank)$;
  if *not v.visited* **then**
    $v.visited \leftarrow true$;
    $successors \leftarrow \{w \in G.successors(v) \mid \text{edge } (v, w) \text{ is not a back edge}\}$;
    if *there exists a back edge* (u, v) *&* $\{w \in successors \mid w \text{ is not post-dominated by } u\} \neq \emptyset$ **then**
      $[loopExit, exitRank] \leftarrow handleLoop()$; /* Find loop exit node/rank */
      $Q.push([loopExit, exitRank])$;
    else if $|successors| > 1$ **then**
      $[mergeNode, mergeRank] \leftarrow handleBranch()$; /* Find node/rank where branch re-joins */
      $Q.push([mergeNode, mergeRank])$;
    end
    forall $s \leftarrow successors$ **do**
      $Q.push([s, rank + 1])$;
    end
  end
end
$contractEmptyRanks()$; /* Remove empty intermediate ranks */

3.1 Layer Assignment

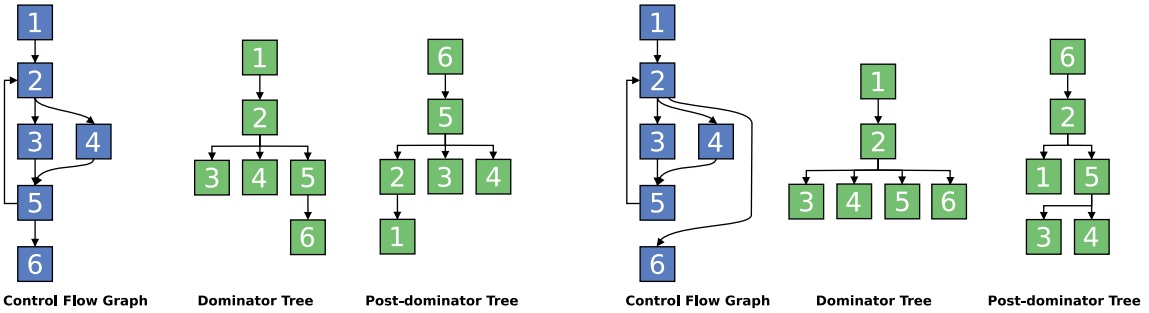
Layer assignment is a crucial phase in layered graph drawing, as it ensures that happens-before relationships are preserved. VEIL's ranking algorithm enforces these relationships by performing a breadth-first traversal of the graph, beginning at the program entry node with an initial rank of $r_0 = 0$. At each subsequent traversal step i, the rank is incremented according to $r_i = r_{i-1} + \delta_r$, where δ_r is 1 for most traversal steps.

When the traversal encounters the start of a loop, δ_r is increased sufficiently so that, by the time the traversal reaches the basic block immediately following the loop at step k, the assigned rank exceeds that of the final node in the loop at step j, i.e., $r_j \leq r_i + \delta_r$. Conversely, when the traversal reaches a conditional split, the algorithm ensures that the rank of the merge point at step k, where the branches meet again, is greater than the maximum rank of all nodes in the conditional branches.

A high level overview of the algorithm is presented in Alg. 1. The handling of loops and conditional branching is discussed in detail in the following Sections 3.1.1 and 3.1.2.

3.1.1 Handling Loops. Control flow graphs may contain three types of loops, illustrated in Fig. 4:

- **Self-loops**. These consist of a single node with an edge pointing back to itself. No special handling is required in our algorithm (Alg. 1), since a breadth-first traversal naturally assigns the exit rank one higher than that of the loop node.



(a) Inverted `do-while` style loop.

(b) Regular `for/while` style loop.

Fig. 4. Examples of regular and inverted loops in CFGs together with their dominator and post-dominator trees.

- **Inverted loops (`do-while` style)**. In these loops, the condition is checked at the end of the body. Again, no special handling is needed: the exit is a successor of the condition check and will be traversed only after the loop body has been ranked. Cases involving branching or `break` statements are covered by the branching rules in subsubsection 3.1.2. An example of an inverted loop's CFG sub-graph can be seen in Fig. 4a.
- **Regular loops (`for/while` style)**. Here, the condition is checked before the body, and special handling is required to correctly identify the loop exit and its rank. An example of this kind of CFG sub-graph can be seen in Fig. 4b, where the condition check, also referred to as the *loop header*, is seen in the node labelled 2.

To detect loops during traversal, we check for incoming back edges. At traversal of node v, if any edge (u, v) with $u \neq v$ is classified as a back edge, then v is part of a loop: either the start of the loop body in an inverted loop, or the loop header in a regular loop. Back edges can be identified efficiently in $O(|V| + |E|)$ through a preprocessing step using depth-first search (DFS) [36].

To distinguish between inverted and regular loops, we examine whether successors of v are post-dominated by u. A node x is *dominated* by a node y, denoted $x \in \text{Dom}(y)$, if all paths from the program entry to x pass through y. Conversely, y *post-dominates* x, denoted $x \in \text{PDom}(y)$, if all paths from x to the program exit pass through y. The sets $\text{Dom}(y)$ and $\text{PDom}(y)$ consequently represent the set of nodes dominated and post-dominated by a node y, respectively. Dominator and post-dominator trees, which are directed graphs where x has an edge to y if x dominates or post-dominates y, respectively, can be precomputed efficiently in effectively linear time through a preprocessing step [12, 59], and are sufficient for answering these queries.

Fig. 4a and Fig. 4b show two examples of an inverted and regular control flow graph together with their corresponding control flow trees. As can be seen in Fig. 4a, all successors of the node 2, which is the target of a back edge, are post-dominated by the source of the back edge, i.e., the node with label 5. This identifies it as an inverted loop. Conversely, in Fig. 4b one successor (node 6) is *not* post-dominated by node 5, and thus this identifies as a regular loop.

Once a regular loop headed at v is detected, we identify the exit node x. This node must be a successor of v, but it cannot be the back edge source u, which lies inside the loop body. Successors of v that dominate u are also excluded, as they, too, lie inside the loop body. If multiple candidates remain, such as when the loop head has several entry paths into the body, the candidate highest in the post-dominator tree (closest to the root) is chosen as the exit node x.

The rank of x is then determined based on the size of the loop body, $|\text{loop}|$. This is computed by counting nodes in a DFS from v, excluding back edges, forward edges, cross edge, and the exit edge (v, x). The exit rank is assigned as

$$x.rank = v.rank + |\text{loop}| + 1$$

and traversal continues at each successor of x.

In loops with extensive branching, the calculation may result in $x.rank \gg v.rank + 1$, leaving empty intermediate layers. These are eliminated in a cleanup pass at the end of the layer assignment phase, which contracts ranks to ensure that only occupied layers remain.

3.1.2 Handling Conditional Splits. A node v visited in the traversal represents a conditional split if it has more than one successor (excluding back edges). For such splits, the merge point can be determined by identifying the immediate post-dominator w of v, i.e., the first successor w of v for which $w \in \text{PDom}(v)$. For example, in the inverted loop in Fig. 4a, the loop body contains a conditional split starting at node 2. A look at the post-dominator tree reveals that the first successor of node 2 which post-dominates, is node 5, which consequently represents the merge point for this split.

To ensure consistent happens-before rankings, the rank of w must be at least one greater than the maximum rank obtained in any of branch of the split. In Fig. 4a, this means that node 5 must be placed below both node 3 and node 4. A direct way to compute this is to consider the difference between the number of nodes dominated by v and those dominated by w. This yields the total number of nodes contained within the conditional structure:

$$|\text{cond}| = |\text{Dom}(v)| - |\text{Dom}(w)|$$

The rank of w is then assigned as

$$w.rank = v.rank + |\text{cond}| + 1$$

As with loops, this calculation may overestimate the necessary distance between v and w, leaving empty ranks, which are removed during cleanup at the end of the layer assignment phase.

3.1.3 Time Complexity. The time complexity of VEIL's layer assignment algorithm is governed by three key steps:

- Ranking is performed in a graph traversal with complexity $O(|V| + |E|)$.
- A pre-processing step needs to identify back edges, which can be done in $O(|V| + |E|)$ time [36].
- Dominator and post-dominator trees need to be computed, which has a worst-case complexity of $O(|V| \cdot \mathcal{T}_d)$, where $\mathcal{T}_d$ is the dominator tree depth. However, in practice this typically takes linear time ($O(|V|)$) for CFGs [12].

By extension, ranking in VEIL has an asymptotic worst-time complexity of $O(|V| \cdot \mathcal{T}_d)$, but is expected to complete in $O(|V| + |E|)$ for most graphs. Even in the worst case, this compares favorably with alternative layout algorithms, such as Dagre [14] or Graphviz's [27] `dot` [32], where typical complexity is $O(|E| \cdot |V| \cdot \log |V|)$.

3.2 Crossing Minimization

In standard layered graph drawing, crossing minimization begins by normalizing edges that span multiple ranks. These edges are subdivided into chains of dummy nodes, reducing the problem to repeated bipartite crossing minimization between the nodes on consecutive ranks V_l and V_{l-1}, with edges E_l spanning the two layers. Since finding the optimal ordering is NP-complete, heuristic methods are used [4, 25, 30, 45]. A widely adopted heuristic is the **barycenter method**, where each node $v \in V_l$ is positioned at the mean of the in-rank positions $u.ord$ of all its successors $u \in suc(v)$:

$$v.ord = \frac{\sum_{w \in suc(v)} w.ord}{|suc(v)|}$$

This heuristic is especially effective on sparse graphs [42, 49], which CFGs are typically a part of. Crossing minimization is typically performed iteratively through alternating upward and downward sweeps of the various bipartite layer graphs until convergence or a cutoff is reached.

VEIL extends this process by keeping back edges intact and treating them explicitly during minimization. When normalizing, back edges are assigned a distinct dummy node type, allowing them to be distinguished from forward edges during reordering. Crossing minimization is then carried out in three steps:

(1) **Pre-sorting**. Each rank is ordered so that dummy nodes from normalized back edges appear first, followed by real nodes, and finally dummy nodes from forward edges. The horizontal position $v.ord$ of each node v within each rank counts as the position for the purpose of the remaining steps.
(2) **Sweep with fixed back edges**. Upward and downward sweeps are performed to minimize crossings, but dummy nodes representing back edges remain fixed at the start of each rank.
(3) **Sweep with movable back edges**. A second sweep is performed where only dummy nodes for back edges are reordered, while all other nodes remain fixed.

This design ensures that back edges are routed consistently to the left side of the layout, while forward edges are biased toward the right. The result is a clearer grouping of long edges (C11; Section 2.2), improving the readability of CFGs.

3.3 Coordinate Assignment

The coordinate assignment phase of VEIL combines the rank assignments and the in-rank ordering information to distribute nodes on a Cartesian grid and then routes edges in straight lines between them. User-configurable spacing values Δ_x and Δ_y define the horizontal and vertical distances between grid points. With coordinates starting at the top left ($x = 0, y = 0$), increasing rightward and downward, a node v in rank $v.rank$ and with in-layer index $v.ord$ is placed as:

$$(v.x, v.y) = (\Delta_x * (v.ord - |BE_{v.rank}|), \Delta_y * v.rank)$$

where $|BE_{v.rank}|$ denotes the number of dummy nodes corresponding to back edges in that layer. This placing forces back edges to be grouped together to the left of the graph, as per C11. However, a user configuration can be used to swap the coordinate assignment function out for $(v.x, v.y) = (\Delta_x * v.ord, \Delta_y * v.rank)$, which instead creates "indentations" in the layout corresponding to loop nest depth, akin to source code.

Once initial positions are assigned, VEIL performs an **edge straightening** pass to align dummy nodes belonging to the same normalized edge on a common x-coordinate. Given a set of dummy nodes V_d representing one normalized edge, the common $x-$coordinate is computed as:

$$x = \begin{cases} \min_{v \in V_d} v.x & \text{if the edge is a back edge,} \\ \max_{v \in V_d} v.x & \text{otherwise.} \end{cases}$$

Edges are then de-normalized, with dummy nodes removed and bends reintroduced at their positions. Optionally, a spline routing step can be applied to smooth out bends before finalizing the layout.

4 Evaluation

To demonstrate the effectiveness of VEIL for control flow graph drawing compared to state-of-the-art approaches, we have implemented VEIL as a layout plugin for the visualization component of the DaCe optimizing compiler [5]. Using this implementation, we compare the resulting layouts for real-world CFGs with those obtained through the default layout algorithm found in the tool, provided through a graph drawing library called Dagre [14]. Dagre implements a version of layered graph drawing with the barycenter heuristic [4, 7, 42] and is used by many web-based graph

visualizations [17–19]. We also compare against Graphviz's [27] `dot` [32] layout, which is one of the most widely used graph drawing algorithms for CFGs, serving as the default for CFGs extracted from Clang/LLVM [44].

4.1 Measuring Aesthetics

To quantitatively evaluate layouts against the established aesthetics criteria, we compute the following metrics:

C1 Node Orthogonality. Efficiency of distributing $|V|$ nodes across the Cartesian grid, scored $\in [0, 1]$:

$$O_V = \frac{|V|}{(w+1)(h+1)}$$

where w and h are the grid dimensions [52].

C2 Edge Orthogonality. Average alignment of edge segments E_s with the x-axis, scored $\in [0, 1]$:

$$O_{E_s} = 1 - \frac{1}{|E_s|} \sum_{e_s \in E_s} \frac{\min(\theta, |90 - \theta|, 180 - \theta)}{45}$$

where θ is the angle of an edge segment to the x-axis [52].

C3 Edge Crossings. Total number of edge crossings per layout.

C4 Edge Bends. Number of intermediate edge points that deviate from a straight line between surrounding points.

C5 Edge Uniformity. Median absolute deviation (MAD) of logarithmic edge lengths in pixels:

$$\text{MAD} = \text{median}(|log(L_i) - log(\tilde{L})|)$$

for the set of edge lengths $L_1, L_2, ..., L_{|E|}$, where $\tilde{L} = \text{median}(L)$.

C6 Short Edges. Total, maximum, and median edge lengths in pixels.

C7 Graph Area. Bounding box area of the graph in square pixels.

C8 Symmetry. Measured via the spring-force tension between nodes [31] (used in force-directed graph drawing algorithms, which inherently optimize for symmetry), reporting the sum and median of all per-node tensions. The per-node tension force $|\vec{F}_v|$ for a node v is given by:

$$|\vec{F}_v| = \left| \underbrace{\sum_{w \in V} \left(\frac{\vec{\Delta}^v_w}{|\vec{\Delta}^v_w|} \cdot \frac{L_U^2}{log(|\vec{\Delta}^v_w|)} \right)}_{\text{repulsive forces}} + \underbrace{\sum_{w \in suc(v)} \left(\frac{\vec{\Delta}^v_w}{|\vec{\Delta}^v_w|} \cdot \frac{log(|\vec{\Delta}^v_w|)^2}{L_U} \right) - \sum_{w \in pred(v)} \left(\frac{\vec{\Delta}^v_w}{|\vec{\Delta}^v_w|} \cdot \frac{log(|\vec{\Delta}^v_w|)^2}{L_U} \right)}_{\text{attractive forces}} \right|$$

where L_U is the ideal edge length, i.e., unit length, the distance in pixels between two ranks, and $\vec{\Delta}^v_w$ is the displacement vector between two nodes v and w.

C9 Consistent Flow. Proportion of edge segments pointing from higher to lower ranks [52], i.e.:

$$\mathcal{CF} = \frac{|\{(u, v) \in E \mid u.rank < v.rank\}|}{|E|}$$

C10 Happens-Before. Since exact evaluation requires topological sorting (computationally infeasible for cyclic graphs), we approximate by measuring the relative rank of the program exit (sink) node to the maximum rank in the graph:

$$\mathcal{HB} = \frac{exit.rank}{|ranks|}$$

C11 Edge Direction Grouping. Median of the minimal distances between any two pairs of back edges or forward edges with overlapping y-coordinates, respectively.

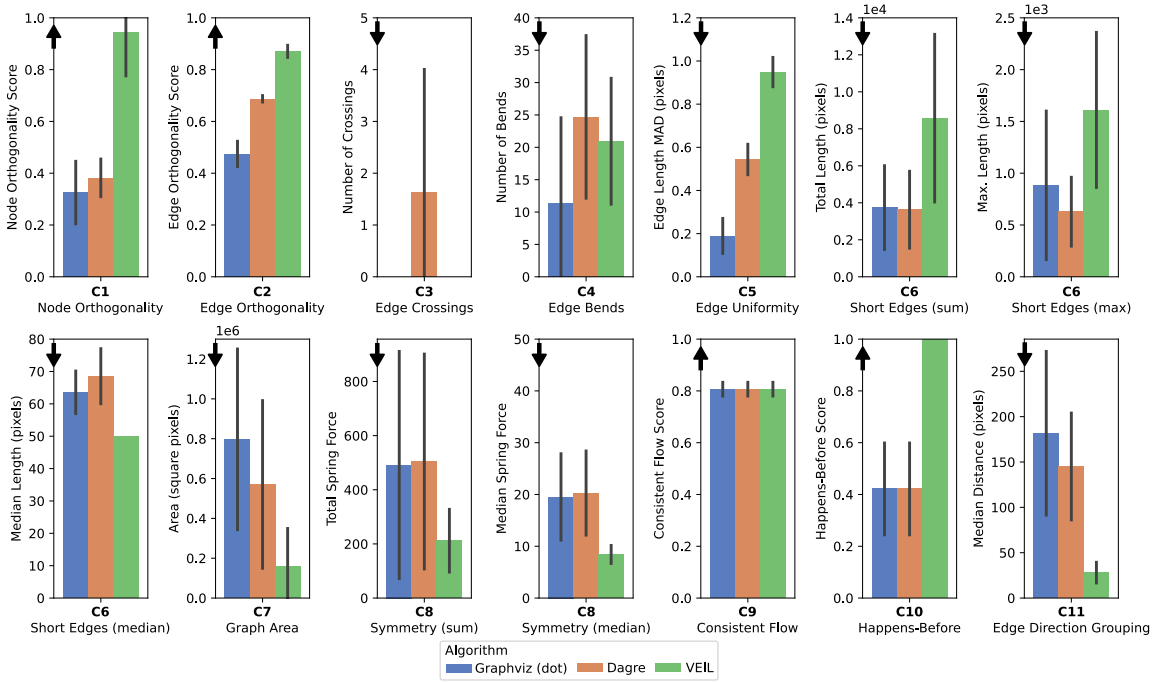


Fig. 5. Criteria metrics across all 30 Polybench applications. Arrows indicate whether higher (↑) or lower (↓) values are better.

For the metrics measuring C1, C2, C9, and C10, higher values are better; for C3–C8 and C11, lower values are better.

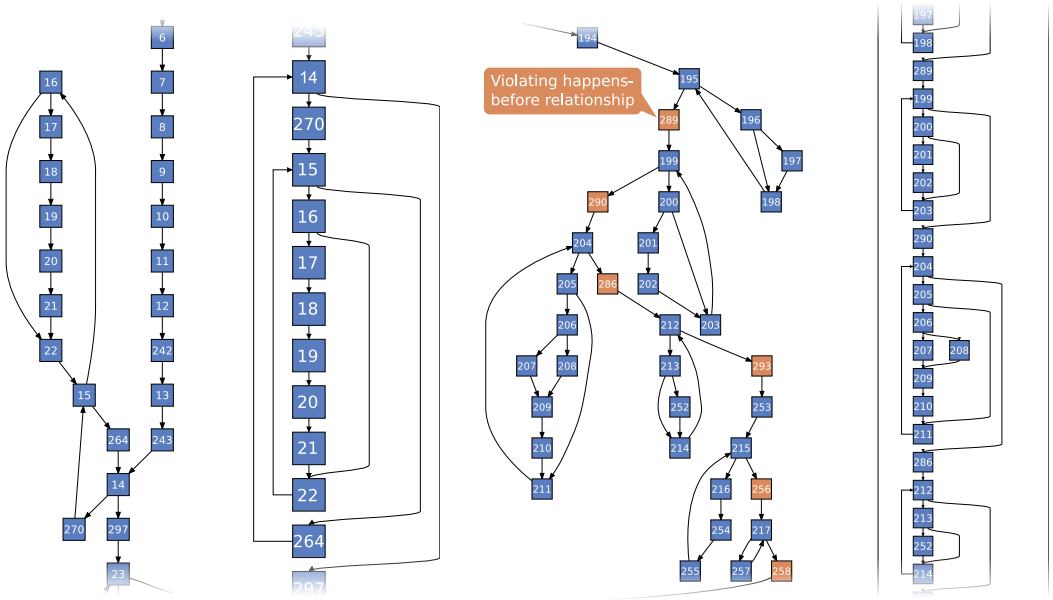
4.2 Polybench

The Polybench/C [51] benchmark suite provides 30 applications representative of program patterns commonly encountered during performance optimization. We use these applications to extract 30 CFGs that are representative of the types of graphs encountered when working with CFGs for compiler development or performance tuning. Each graph is drawn using VEIL, Dagre [14], and Graphviz's `dot` [27, 32]. For each layout, we compute the metrics described in Section 4.1. Fig. 5 summarizes the results across all 30 CFGs, reporting median values with standard deviation error bars.

For node and edge orthogonality (C1, C2), VEIL achieves significantly higher scores than both Dagre and `dot`. In edge crossings (C3), VEIL and `dot` consistently avoid crossings, while Dagre does not. The number of edge bends (C4) is comparable across all algorithms. However, in VEIL, bends correspond directly to meaningful control-flow constructs, such as loops or conditional skips.

As expected from enforcing happens-before relationships, VEIL exhibits higher edge length variability (C5) and longer total and maximum edge length (C6) compared to the baselines. Yet, VEIL achieves a lower median edge length with zero variance, indicating that the majority of edges are unit-length (spanning a single rank). Similar to edge bends, longer edges in VEIL represent semantically important control flow constructs.

VEIL also outperforms the baselines on graph area (C7) and symmetry (C8), reducing area by up to **5**× and global spring force by 2.4× compared to the baselines. Consistent flow (C9) is identical across all three algorithms, while VEIL uniquely guarantees a perfect happens-before score (C10), representing a 2.4× improvement over Dagre and



(a) First loop nest (dot) (b) First loop nest (VEIL) (c) dot: Orange nodes violate C10. (d) VEIL: No C10 violations.

Fig. 6. Various loop nests found in CLOUDSC demonstrate how existing approaches (dot) fail to respect happens-before relationships and edge direction grouping, decreasing readability. VEIL drawings of the same loop nests offer improved readability.

dot. Additionally, similar edge types in VEIL are more tightly grouped (C11), leading to an up to 7.4× reduction in the distance between similar edge types compared to the baselines.

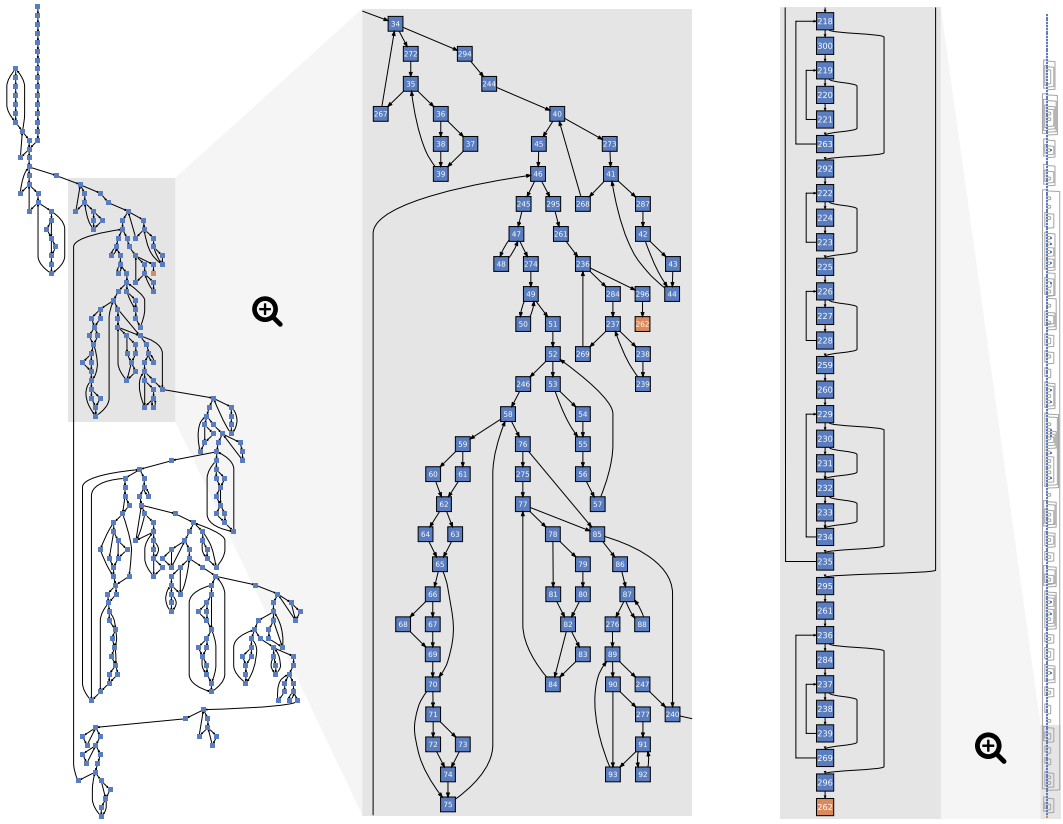
The improved algorithmic complexity of VEIL's layer assignment phase compared to Dagre and dot additionally leads to an average of 1.4× faster layout times than measured in Dagre.

4.3 CLOUDSC

The European Centre for Medium-Range Weather Forecasts' (ECMWF) Integrated Forecasting System (IFS) contains a physics simulation, the cloud microphysics scheme (CLOUDSC) [28], which exhibits complex control flow patterns typical for such simulations. This class of applications frequently features deep nested loops, state-dependent branching (i.e,. large, different code paths), early termination, or error correction. Such patterns often produce layouts that violate happens-before relationships, making CFGs difficult to interpret.

Fig. 6a illustrates this problem in a CFG section drawn with Graphviz/dot. After an initial linear sequence of nodes, the flow unexpectedly turns upwards, introducing two consecutive upward edges. While loops are present, the upward edges do not correspond to back edges but to loop bodies, obscuring their structure. In contrast, VEIL (Fig. 6b) correctly identifies loop bodies and exits using dominator analysis, arranges nodes in proper topological order, and routes only back edges upwards. Edge direction grouping (C11) further improves readability by keeping back edge consistently on the left, revealing loop nesting, while skip edges are kept right to highlight the conditional execution of the entire loop body.

A similar issue is shown in Fig. 6c, where loop exit nodes appear above their corresponding loop bodies in the dot layout, violating happens-before and hindering loop identification. VEIL (Fig. 6d) places exit nodes below their bodies



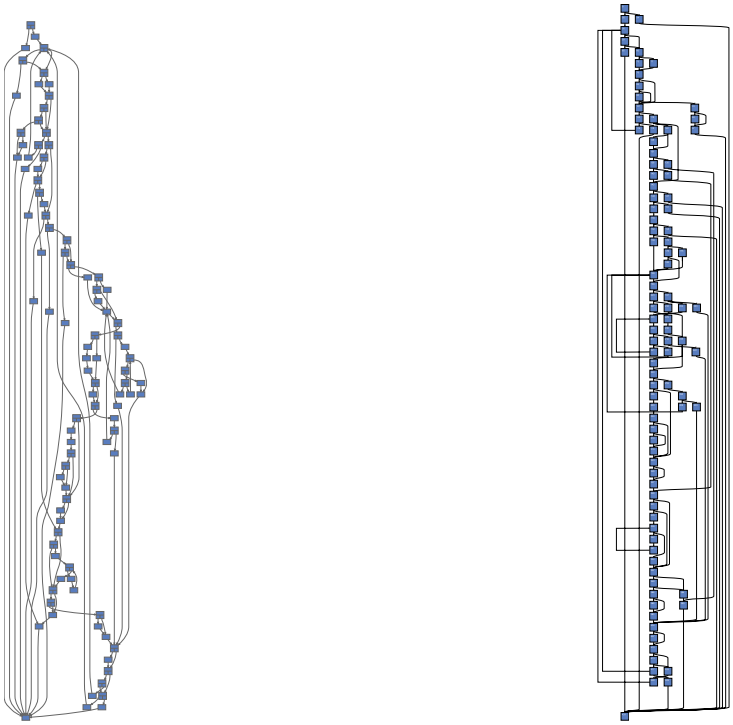
(a) Graphviz's `dot` places the program exit (orange) close to the top of the graph. (b) VEIL keeps the program exit at the bottom.

Fig. 7. Established layouts violate happens-before relationships (C10) by drawing CLOUDSC's program exit (orange) close to the top.

and groups back edges on the left, making nested loops and their extents immediately visible. The presence of a second, long back edge on the far left even exposes the existence of a larger enclosing loop, which is not discernible in the `dot` layout.

Fig. 7a shows another example, where `dot` places the program exit block in the upper third of the layout, surrounded by other nodes. Locating the program exit and early terminations leading to it thus requires manually scanning for sink nodes, a task that becomes infeasible in graphs with thousands of nodes. VEIL (Fig. 7b) instead places the exit at the bottom of the layout, making program termination trivial to locate and clearly showing a final nested loop which each program exit path runs through.

4.3.1 Quantitative Comparison. Quantitatively, VEIL achieves up to **10**× higher node orthogonality (C1) and 1.7× higher edge orthogonality (C2) than Dagre and `dot`, along with a 2.5× reduction in edge bends (C4). While `dot` produces slightly more uniform and shorter edges (C5, C6), VEIL emphasizes unit-length edges and leverages longer edges to highlight loops and skips (cf. Fig. 6b). Median edge length in VEIL is up to 1.8× shorter than in the baselines, underscoring its bias toward compact local flow. VEIL also consistently achieves a happens-before score (C10) of 1, representing a 3× improvement over the baselines for CLOUDSC. Thanks to the reduced ranking time complexity, VEIL



(a) Graphviz's layout leads to mixing of back edges and forward edges, making control flow paths formed by gotos hard to trace.

(b) Through VEIL's edge direction grouping (C11), the quasi-loop structures formed by gotos are exposed.

Fig. 8. The CFG of Linux's shmem_getpage_gfp memory page allocation function, drawn via Graphviz and VEIL.

additionally completes the layout 1.6× faster than Dagre. Finally, the resulting slender, vertical layout reduces overall graph area (C7) by up to **13.1**×.

4.4 Linux Kernel Memory Management

To demonstrate how VEIL helps to interpret irreducible / irregular control flow, we analyze the `shmem_getpage_gfp` function inside the Linux [64] operating system. The function is designed to find a memory page in cache or swap, or allocate it if can not be found. As with much of the Linux kernel code, the function is characterized by a number of `goto` statements (12 `goto` statements and 6 labels) which form unstructured / irreducible control flow. The function also contains some conditional branching, early return statements, and loops. As such, even the code can be difficult to analyze, and the resulting CFG layout in Graphviz's `dot` becomes difficult to read, with edge crossings, a large number of upwards running edges, and many long, ungrouped edges (Fig. 8a).

Through VEIL's grouping of back edges, control flow becomes more readable and the back edges indicate how `goto` statements in the loop serve to form quasi-loops. Similarly, various long skip edges routed to the right indicate multiple early return paths, while illustrating how `goto` statements are used to run error handling before exiting. Analysis of codes containing such irregular control flow constructs is difficult even in code, which makes both debugging and security vulnerability analysis of such codes challenging. Consequently, edge grouping helps for a mental map by more clearly showing repeat or skipping behavior, helping in tracing possible execution paths through the program.

5 Related Work

Graph drawing and software visualization are both highly active research areas. We discuss a few of the advances and techniques most relevant or closely related to the contributions made in this paper, and discuss how VEIL separates itself from them.

General Graph Drawing. Numerous general graph drawing algorithms have been developed to improve the readability of complex and large graphs. The Sugiyama Method [60], or layered graph drawing, has become the most popular framework for drawing directed acyclic graphs, due to the readability benefits obtained by its strict enforcement of a consistent flow direction. Many state-of-the art algorithms build on that framework, most notably Graphviz [27] using the `dot` [32] algorithm, the VCG tool [58], or Dagre [14]. Various improvements have been made to different parts of layered graph drawing, including novel heuristic methods for reducing edge crossings [25, 42, 48].

Dig-CoLa [22, 23] combines layered drawing with force directed layout techniques to improve edge direction uniformity for directed graphs. However, the technique requires graphs to be acyclic, leading to similar problems with CFGs as traditional layered graph drawing algorithms.

IPSep-CoLa [24] goes in a similar direction, allowing for certain relative placement constraints between node pairs to be introduced. Automatically deducing such constraints to enforce control flow-specific aesthetics criteria is challenging, making the technique less suitable for automatic CFG drawing.

Carmel et al. [10] introduced a technique where force minimization is combined with layered graph drawing, enabling drawing for both cyclic and acyclic graphs by not restricting nodes to strict ranks. The approach is well suited for higher degree graphs, and does not aim to strictly enforce happens-before relationships, therefore being less suitable for control flow graphs.

Control Flow Graph Drawing. Not many works have focused on improving graph drawing specifically for control flow graphs, but a few techniques have been put forth.

CFGExplorer [18], CFGConf [19], and CcNav [17] have addressed the issue of happens-before relationships being violated after with loops by forcing layered graph drawing algorithms to assign specific ranks to loop exits through invisible edges inserted during ranking. While this improves on the happens-before criterion for loops, the remaining layout still relies on Dagre. VEIL further improves on this by providing edge grouping and adding semantic meaning to edge lengths and bends.

`regVIS` [63] uses regular expression-based parsing of control flow graphs to extract common control flow constructs such as loops and visualize them in more intuitive manners, largely avoiding CFG layout issues. The reliance on regular expressions means that the technique primarily handles regular / reducible control flow constructs.

Robillard and Simoneau [57] have developed an alternative approach of visualizing control flow graphs using an iconic representation. The technique avoids many layout related issues and produces slender, vertical control flow graph visualizations similar to VEIL, however with less strict adherence to other established graph aesthetics criteria.

In a direction similar to control flow graphs, Würthinger et al. [69] have developed a system for visualizing program dependence graphs, which show program dependency relationships similar to control flow dependencies. Balmas [3] investigated a similar system for visualizing system dependence graphs. Under the hood, both systems rely on Graphviz's `dot` layout, meaning cycles in control flow graphs lead to similar violations of happens-before relationships demonstrated in this paper.

6 Conclusion

We present VEIL, a novel graph drawing algorithm for control flow graphs that explicitly incorporates program semantics into the drawing process. Unlike general-purpose graph drawing approaches, VEIL leverages dominator analysis and CFG-specific aesthetics criteria to produce layouts that better reflect execution order and structural program constructs. In doing so, it codifies and extends established graph drawing aesthetics with two new domain-specific criteria: happens-before ordering (C10) and edge direction grouping (C11).

Through a systematic evaluation on real-world CFGs, we compare VEIL against state-of-the-art baselines in layered graph drawing across 11 quantitative metrics. The results demonstrate that VEIL consistently improves adherence to CFG-relevant layout criteria while maintaining or exceeding performance on general-purpose graph drawing aesthetics. In particular, VEIL provides clearer visual cues for loops, branching, and execution flow, thereby improving readability and enabling users to reason about program behavior more naturally, akin to reading source code.

Looking ahead, our work suggests several promising directions. Integration of VEIL into mainstream compiler toolchains and security analysis environments could enhance the usability of CFG visualizations in practice. Further user studies would also complement our quantitative evaluation by assessing human comprehension directly. Finally, the principles underlying VEIL may extend beyond CFGs to other hierarchical graph domains where execution order or semantic flow is central.

In summary, VEIL demonstrates that domain-aware layout algorithms can substantially advance the clarity and effectiveness of program visualizations, bridging the gap between formal graph structures and human program understanding.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 program (grant agreement PSAP, No. 101002047), and from the European High Performance Computing Joint Undertaking (EuroHPC-JU) under the DEEP-SEA program (grant agreement No. 955606). Work by Lawrence Livermore National Laboratory was performed under the auspices of the U.S. Department of Energy under contract DE-AC52-07NA27344 (LLNL-CONF-2013055).

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security* 13, 1 (Oct. 2009), 1–40. doi:10.1145/1609956.1609960

[2] Shahid Alam, Issa Traore, and Ibrahim Sogukpinar. 2015. Annotated Control Flow Graph for Metamorphic Malware Detection. *Comput. J.* 58, 10 (Oct. 2015), 2608–2621. doi:10.1093/comjnl/bxu148

[3] Francoise Balmas. 2004. Displaying dependence graphs: a hierarchical approach. *Journal of Software Maintenance and Evolution: Research and Practice* 16, 3 (May 2004), 151–185. doi:10.1002/smr.291

[4] Wilhelm Barth, Michael Jünger, and Petra Mutzel. 2002. Simple and Efficient Bilayer Cross Counting. In *Graph Drawing*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Michael T. Goodrich, and Stephen G. Kobourov (Eds.). Vol. 2528. Springer Berlin Heidelberg, Berlin, Heidelberg, 130–141. doi:10.1007/3-540-36151-0_13 Series Title: Lecture Notes in Computer Science.

[5] Tal Ben-Nun, Johannes De Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: a data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Denver Colorado, 1–14. doi:10.1145/3295500.3356173

[6] Chris Bennett, Jody Ryall, Leo Spalteholz, and Amy Gooch. 2007. The Aesthetics of Graph Visualization. In *Proceedings of the Eurographics Workshop on Computational Aesthetics in Graphics, Visualization and Imaging*. Banff, Alberta, Canada.

[7] Ulrik Brandes and Boris Köpf. 2002. Fast and Simple Horizontal Coordinate Assignment. In *Graph Drawing*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Petra Mutzel, Michael Jünger, and Sebastian Leipert (Eds.). Vol. 2265. Springer Berlin Heidelberg, Berlin, Heidelberg, 31–44. doi:10.1007/3-540-45848-4_3 Series Title: Lecture Notes in Computer Science.

[8] Michael Burch. 2017. A User Study on Judging the Target Node in Partial Link Drawings. In *2017 21st International Conference Information Visualisation (IV)*. IEEE, London, 199–204. doi:10.1109/iV.2017.43

[9] Michael Burch, Weidong Huang, Mathew Wakefield, Helen C. Purchase, Daniel Weiskopf, and Jie Hua. 2021. The State of the Art in Empirical User Evaluation of Graph Visualizations. *IEEE Access* 9 (2021), 4173–4198. doi:10.1109/ACCESS.2020.3047616

[10] Liran Carmel, David Harel, and Yehuda Koren. 2004. Combining hierarchy and energy for drawing directed graphs. *IEEE Transactions on Visualization and Computer Graphics* 10, 1 (Jan. 2004), 46–57. doi:10.1109/TVCG.2004.1260757

[11] Compiler Explorer LLC. 2025. *Compiler Explorer*. https://compiler-explorer.com/

[12] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. 2006. *A Simple, Fast Dominance Algorithm*. Technical Report TR-06-33870. Rice University.

[13] Chris Cummins, Zacharias V Fisches, Tal Ben-Nun, Torsten Hoefler, Michael O'Boyle, and Hugh Leather. 2021. PROGRAML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proceedings of the 38th International Conference on Machine Learning*, Vol. 139. 2244–2253.

[14] dagrejs. 2025. *Directed graph layout for JavaScript*. https://github.com/dagrejs/dagre

[15] Ron Davidson and David Harel. 1996. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics* 15, 4 (Oct. 1996), 301–331. doi:10.1145/234535.234538

[16] Felice De Luca, Stephen Kobourov, and Helen Purchase. 2018. Perception of Symmetries in Drawings of Graphs. In *Graph Drawing and Network Visualization*, Therese Biedl and Andreas Kerren (Eds.). Vol. 11282. Springer International Publishing, Cham, 433–446. doi:10.1007/978-3-030-04414-5_31 Series Title: Lecture Notes in Computer Science.

[17] Sabin Devkota, Pascal Aschwanden, Adam Kunen, Matthew Legendre, and Katherine E. Isaacs. 2021. CcNav: Understanding Compiler Optimizations in Binary Code. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (Feb. 2021), 667–677. doi:10.1109/TVCG.2020.3030357

[18] Sabin Devkota and Katherine E. Isaacs. 2018. CFGExplorer: Designing a Visual Control Flow Analytics System around Basic Program Analysis Operations. *Computer Graphics Forum* 37, 3 (June 2018), 453–464. doi:10.1111/cgf.13433

[19] Sabin Devkota, Matthew P. LeGendre, Adam Kunen, Pascal Aschwanden, and Katherine E. Isaacs. 2022. Domain-Centered Support for Layout, Tasks, and Specification for Control Flow Graph Visualization. In *2022 Working Conference on Software Visualization (VISSOFT)*. IEEE, Limassol, Cyprus, 40–50. doi:10.1109/VISSOFT55257.2022.00013

[20] Walter Didimo, Evgenios M. Kornaropoulos, Fabrizio Montecchiani, and Ioannis G. Tollis. 2018. A Visualization Framework and User Studies for Overloaded Orthogonal Drawings. *Computer Graphics Forum* 37, 1 (Feb. 2018), 288–300. doi:10.1111/cgf.13266

[21] Bradley Dux, Anand Iyer, Saumya Debray, David Forrester, and Stephen Kobourov. 2005. Visualizing the Behavior of Dynamically Modifiable Code. In *13th International Workshop on Program Comprehension (IWPC'05)*. IEEE, St. Louis, MO, USA, 337–340. doi:10.1109/WPC.2005.45

[22] Tim Dwyer and Yehuda Koren. 2005. Dig-CoLa: directed graph layout through constrained energy minimization. In *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.* IEEE, Minneapolis, MN, USA, 65–72. doi:10.1109/INFVIS.2005.1532130

[23] Tim Dwyer, Yehuda Koren, and Kim Marriott. 2006. Drawing directed graphs using quadratic programming. *IEEE Transactions on Visualization and Computer Graphics* 12, 4 (July 2006), 536–548. doi:10.1109/TVCG.2006.67

[24] Tim Dwyer, Yehuda Koren, and Kim Marriott. 2006. IPSep-CoLa: An Incremental Procedure for Separation Constraint Layout of Graphs. *IEEE Transactions on Visualization and Computer Graphics* 12, 5 (Sept. 2006), 821–828. doi:10.1109/TVCG.2006.156

[25] Peter Eades and Nicholas C. Wormald. 1994. Edge crossings in drawings of bipartite graphs. *Algorithmica* 11, 4 (April 1994), 379–403. doi:10.1007/BF01187020

[26] Holger Eichelberger. 2002. Aesthetics of class diagrams. In *Proceedings First International Workshop on Visualizing Software for Understanding and Analysis*. IEEE Comput. Soc, Paris, France, 23–31. doi:10.1109/VISSOF.2002.1019791

[27] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. 2002. Graphviz— Open Source Graph Drawing Tools. In *Graph Drawing*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Petra Mutzel, Michael Jünger, and Sebastian Leipert (Eds.). Vol. 2265. Springer Berlin Heidelberg, Berlin, Heidelberg, 483–484. doi:10.1007/3-540-45848-4_57 Series Title: Lecture Notes in Computer Science.

[28] European Centre for Medium-Range Weather Forecasts. 2003. *CLOUDSC cloud microphysics scheme*. https://github.com/ecmwf-ifs/dwarf-p-cloudsc

[29] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (July 1987), 319–349. doi:10.1145/24039.24041

[30] Michael Forster. 2005. A Fast and Simple Heuristic for Constrained Two-Level Crossing Reduction. In *Graph Drawing*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, and János Pach (Eds.). Vol. 3383. Springer Berlin Heidelberg, Berlin, Heidelberg, 206–216. doi:10.1007/978-3-540-31843-9_22 Series Title: Lecture Notes in Computer Science.

[31] Thomas M. J. Fruchterman and Edward M. Reingold. 1991. Graph drawing by force-directed placement. *Software: Practice and Experience* 21, 11 (Nov. 1991), 1129–1164. doi:10.1002/spe.4380211102

[32] Emden Gansner, Eleftherios Koutsofios, and Stephen North. 2015. *Drawing graphs with dot*. Technical Report.

[33] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. 1993. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering* 19, 3 (March 1993), 214–230. doi:10.1109/32.221135

[34] Michael R. Garey and David S. Johnson. 1983. Crossing Number is NP-Complete. *SIAM Journal on Algebraic Discrete Methods* 4, 3 (Sept. 1983), 312–316. doi:10.1137/0604033

[35] Helen Gibson, Joe Faith, and Paul Vickers. 2013. A survey of two-dimensional graph layout techniques for information visualisation. *Information Visualization* 12, 3-4 (July 2013), 324–357. doi:10.1177/1473871612455749

[36] Matthew S. Hecht and Jeffrey D. Ullman. 1974. Characterizations of Reducible Flow Graphs. *J. ACM* 21, 3 (July 1974), 367–375. doi:10.1145/321832.321835

[37] Weidong Huang. 2013. Establishing aesthetics based on human graph reading behavior: two eye tracking studies. *Personal and Ubiquitous Computing* 17, 1 (Jan. 2013), 93–105. doi:10.1007/s00779-011-0473-2

[38] Weidong Huang, Peter Eades, and Seok-Hee Hong. 2005. *Layout effects: comparison of sociogram drawing conventions.* School of Information Technologies, University of Sydney, Sydney. OCLC: 225350792.

[39] Weidong Huang and Maolin Huang. 2010. Exploring the relative importance of crossing number and crossing angle. In *Proceedings of the 3rd International Symposium on Visual Information Communication*. ACM, Beijing China, 1–8. doi:10.1145/1865841.1865854

[40] Weidong Huang, Mao Lin Huang, and Chun-Cheng Lin. 2016. Evaluating overall quality of graph visualizations based on aesthetics aggregation. *Information Sciences* 330 (Feb. 2016), 444–454. doi:10.1016/j.ins.2015.05.028

[41] Weidong Huang, Chun-Cheng Lin, and Mao Lin Huang. 2012. An aggregation-based approach to quality evaluation of graph drawings. In *Proceedings of the 5th International Symposium on Visual Information Communication and Interaction*. ACM, Hangzhou China, 110–113. doi:10.1145/2397696.2397712

[42] Michael Jünger and Petra Mutzel. 2002. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. In *Graph algorithms and applications i*. World Scientific, 3–27.

[43] Rainer Koschke. 2002. Software Visualization for Reverse Engineering. In *Software Visualization*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, and Stephan Diehl (Eds.). Vol. 2269. Springer Berlin Heidelberg, Berlin, Heidelberg, 138–150. doi:10.1007/3-540-45875-1_11 Series Title: Lecture Notes in Computer Science.

[44] Chris Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, San Jose, CA, USA, 75–86. doi:10.1109/CGO.2004.1281665

[45] Xiao Yu Li and Matthias F. Stallmann. 2002. New bounds on the barycenter heuristic for bipartite graph drawing. *Inform. Process. Lett.* 82, 6 (June 2002), 293–298. doi:10.1016/S0020-0190(01)00297-6

[46] William Mansky, Elsa L. Gunter, Dennis Griffith, and Michael D. Adams. 2016. Specifying and executing optimizations for generalized control flow graphs. *Science of Computer Programming* 130 (Nov. 2016), 2–23. doi:10.1016/j.scico.2016.06.003

[47] Kazuo Misue. 2008. Anchored Map: Graph Drawing Technique to Support Network Mining. *IEICE Transactions on Information and Systems* E91-D, 11 (Nov. 2008), 2599–2606. doi:10.1093/ietisy/e91-d.11.2599

[48] Petra Mutzel. 1997. An alternative method to crossing minimization on hierarchical graphs: Extended abstract. In *Graph Drawing*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, and Stephen North (Eds.). Vol. 1190. Springer Berlin Heidelberg, Berlin, Heidelberg, 318–333. doi:10.1007/3-540-62495-3_57 Series Title: Lecture Notes in Computer Science.

[49] Erkki Mäkinen. 1990. Experiments on drawing 2-level hierarchical graphs. *International Journal of Computer Mathematics* 36, 3-4 (Jan. 1990), 175–181. doi:10.1080/00207169008803921

[50] Object Management Group Standards Development Organization. 2025. *UML Unified Modeling Language.* https://www.uml.org/

[51] Louis-Noël Pouchet, Tomofumi Yuki, and Matthias J. Reisinger. 2025. *PolyBench/C 4.2.1 (beta).* https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1

[52] Helen C. Purchase. 2002. Metrics for Graph Drawing Aesthetics. *Journal of Visual Languages & Computing* 13, 5 (Oct. 2002), 501–516. doi:10.1006/jvlc.2002.0232

[53] Helen C. Purchase, David Carrington, and Jo-Anne Allder. 2002. Empirical Evaluation of Aesthetics-based Graph Layout. *Empirical Software Engineering* 7, 3 (Sept. 2002), 233–255. doi:10.1023/A:1016344215610

[54] Helen C. Purchase, Robert F. Cohen, and Murray I. James. 1997. An experimental study of the basis for graph drawing algorithms. *ACM Journal of Experimental Algorithmics* 2 (Jan. 1997), 4. doi:10.1145/264216.264222

[55] Helen C Purchase, Matthew McGill, Linda Colpoys, and David Carrington. 2001. Graph drawing aesthetics and the comprehension of UML class diagrams: an empirical study, Vol. 9. https://dl.acm.org/doi/abs/10.5555/564040.564056

[56] Matthew Revelle, Matt Parker, and Kevin Orr. 2023. Blaze: A Framework for Interprocedural Binary Analysis. In *Proceedings 2023 Workshop on Binary Analysis Research*. Internet Society, San Diego, CA, USA. doi:10.14722/bar.2023.23009

[57] Pierre N. Robillard and Mario Simoneau. 1993. Iconic control graph representation. *Software: Practice and Experience* 23, 2 (Feb. 1993), 223–234. doi:10.1002/spe.4380230206

[58] Georg Sander. 1995. Graph layout through the VCG tool: Extended abstract and system demonstration. In *Graph Drawing*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Roberto Tamassia, and Ioannis G. Tollis (Eds.). Vol. 894. Springer Berlin Heidelberg, Berlin, Heidelberg, 194–205. doi:10.1007/3-540-58950-3_371 Series Title: Lecture Notes in Computer Science.

[59] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. 1997. Incremental computation of dominator trees. *ACM Transactions on Programming Languages and Systems* 19, 2 (March 1997), 239–252. doi:10.1145/244795.244799

[60] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. 1981. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics* 11, 2 (1981), 109–125. doi:10.1109/TSMC.1981.4308636

[61] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. 1988. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man, and Cybernetics* 18, 1 (Feb. 1988), 61–79. doi:10.1109/21.87055

[62] Martyn Taylor and Peter Rodgers. 2005. Applying Graphical Design Techniques to Graph Visualisation. In *Ninth International Conference on Information Visualisation (IV'05)*. IEEE, London, England, 651–656. doi:10.1109/IV.2005.19

[63] Sibel Toprak, Arne Wichmann, and Sibylle Schupp. 2014. Lightweight Structured Visualization of Assembler Control Flow Based on Regular Expressions. In *2014 Second IEEE Working Conference on Software Visualization*. IEEE, Victoria, BC, Canada, 97–106. doi:10.1109/VISSOFT.2014.25

[64] Linus Torvalds. 2025. *Linux kernel source tree*. https://github.com/torvalds/linux

[65] Yong Wang, Qiaomu Shen, Daniel Archambault, Zhiguang Zhou, Min Zhu, Sixiao Yang, and Huamin Qu. 2016. AmbiguityVis: Visualization of Ambiguity in Graph Layouts. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (Jan. 2016), 359–368. doi:10.1109/TVCG.2015.2467691

[66] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. 2002. Cognitive Measurements of Graph Aesthetics. *Information Visualization* 1, 2 (June 2002), 103–110. doi:10.1057/palgrave.ivs.9500013

[67] Eric Welch and Stephen Kobourov. 2017. Measuring Symmetry in Drawings of Graphs. *Computer Graphics Forum* 36, 3 (June 2017), 341–351. doi:10.1111/cgf.13192

[68] Kenny Wong and Dabo Sun. 2006. On evaluating the layout of UML diagrams for program comprehension. *Software Quality Journal* 14, 3 (Sept. 2006), 233–259. doi:10.1007/s11219-006-9218-2

[69] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. 2008. Visualization of Program Dependence Graphs. In *Compiler Construction*, Laurie Hendren (Ed.). Vol. 4959. Springer Berlin Heidelberg, Berlin, Heidelberg, 193–196. doi:10.1007/978-3-540-78791-4_13 Series Title: Lecture Notes in Computer Science.