# Towards Better CFG Layouts

Jack Royer[‡*], Frédéric Tronel[‡§], Yaëlle Vinçont[†¶]
[‡]CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA
[†]Univ Rennes, Inria, CNRS, IRISA
[*]jack.royer@centralesupelec.fr
[§]frederic.tronel@centralesupelec.fr
[¶]yaelle.vincont@inria.fr

***Abstract*—Reverse engineering of software is used to analyze the behavior of malicious programs, find vulnerabilities in software, or design interoperability solutions. Although this activity largely relies on dedicated software toolbox, it is still largely manual. In order to facilitate these tasks, many tools provide analysts with an interface to visualize Control Flow Graph (CFG) of a function. Properly laying out the CFG is therefore extremely important to facilitate manual reverse engineering. However, CFGs are often laid out with general algorithms rather than domain-specific ones. This leads to subpar graph layouts. In this paper, we provide a comprehensive state-of-the-art for CFG layout techniques. We propose a modified layout algorithm that showcases the patterns analysts are looking for. Finally, we compare layouts offered by popular binary analysis frameworks with our own.**

## I. Introduction

A Control Flow Graph (CFG) is a visual representation of the different execution paths of a program. These graphs are widely used in binary analysis, with many binary analysis programs [1], [7], [10], [13] allowing the user to interact with a program's CFG. Although domain-specific visualization tools exist, such as CFGConf [4], many binary analysis tools still use the generic layered graph drawing technique described by Sugiyama [14] in 1985. Such is the case of `rev.ng` [12], according to private communications with the authors.

Through this paper, we hope to shed some light on the considerations taken into account when designing a CFG layout and compare real world binary analysis programs. We hope that this can lead to more transparency when it comes to laying out CFGs and better layouts for all. To this end, we will publish our sources, and highly encourage future discussions on CFG layouts.

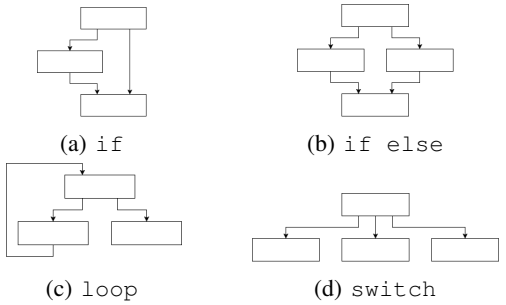
In this paper, we: 1) propose a *domain specific algorithm* for CFG layouts based on Single Entry Single Exit (SESE) regions [8]; 2) *compare this new layout* with the layouts offered by various binary analysis tools; 3) discuss the *challenges* faced when making such comparisons; 4) offer a *new way of evaluating CFG layout algorithms* by using CFGs provided by LLVM.

Workshop on Binary Analysis Research (BAR) 2025
28 February 2025, San Diego, CA, USA
ISBN 979-8-9919276-4-2
https://dx.doi.org/10.14722/bar.2025.23011
www.ndss-symposium.org

## II. Context

Reverse engineering frameworks often allow analysts to read assembly in a *graph view*. In this mode, instructions are not displayed linearly but rather inside a 2D graph: the Control Flow Graph.

This representation helps an analyst identify common patterns in the assembly code such as loops, if statements, etc. Figure 1 contains some of these patterns. Without this representation, the analyst would have to manually untangle the blocks based on the control flow instructions.



(a) `if` (b) `if else` (c) `loop` (d) `switch`

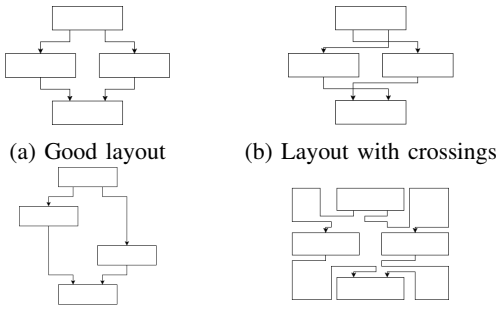
Fig. 1: Common, visually identifiable, patterns in CFGs

It is therefore no surprise that this representation is extremely prominent in reverse engineering tools. As a matter of fact, a similar *graph view* exists in major binary analysis tools such as IDA [7], Ghidra [10], Binary Ninja [1], `rev.ng` [12] and Iato [11].

### A. Graph layout metrics

To maximize the knowledge gained from this representation, it is important that the graph be laid out as clearly as possible. However, clarity is not an objective criteria, and thus not easy to measure. Instead, here are a couple of metrics which can serve as indicators of the clarity of a graph's layout:

- **Edge crossings**: fewer edge crossings facilitate the comprehension of relationships within the graph;
- **Compactness**: smaller, more compact graphs are often preferable to larger ones with a lot of empty space;
- **Edge straightness**: simple straight edges are easier to follow than crooked ones.



(a) Good layout (b) Layout with crossings

(c) Tall layout (d) Wobbly edges

Fig. 2: Imperfections in graph layout

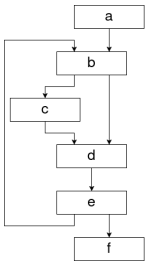
Figure 2 illustrates the impacts of these metrics on the clarity of a graph's layout.

### B. Additional CFG specific layout constraints

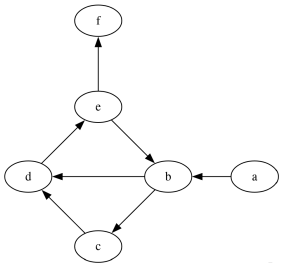
Laying out CFGs comes with an additional constraint: it is important that the layout makes *semantic sense*. Using `graphviz`'s `circo` engine to display the CFG of the program shown in fig. 3a, one can obtain the graph in figure fig. 3c. This layout scores highly with the previously described metrics: the edges are short, there are no crossings, and the graph is compact. However, it is very difficult to understand what program this CFG represents. With a little reorganization, as done in fig. 3b, we can see that this program contains a `do while` style loop with an `if` statement in the loop.

```
a
do
    if b then
        c
    end if
    d
while e
f
```

(a) Example program




(b) Corresponding CFG, manually laid out




(c) Same graph, different layout (using `graphviz circo`)

```
digraph G {
    c -> d
    d -> e
    e -> f
    e -> b
    b -> d
    b -> c
    a -> b
}
```

(d) Code for the `graphviz circo` graph

Fig. 3: Impact of graph layout on the clarity of a CFG

## III. Related work

According to private communications initiated by the authors with several binary reverse engineering software maintainers, and work done by Sabin Devkota *et al.* [5], CFGs are often laid out using *layered graph drawing*. Described in Sugiyama *et al.*'s paper *Methods for visual understanding of hierarchical system structures* [14], this algorithm creates the structures analysts are looking for in CFGs.

### A. Layered Graph Drawing

The Sugiyama algorithm breaks down graph layout into multiple smaller problems.

*1) Cycle removal:* During this step, cycles are (temporarily) removed from the graph, facilitating the next steps.

*2) Layer assignment:* During layer assignment, each node is attributed a layer in order to mimic the execution order of nodes. Some edges might span multiples layers, for example if an edge connects a node from layer 0 to one from node 2. Such edges are called *long edges*. These edges are split by creating additional *dummy nodes*.

*3) Node ordering:* Once each node has been assigned a layer, nodes of the same layer are permuted in order to reduce the amount of edge crossings.

*4) Coordinate assignment:* The final step consists in transforming layers and cycles into coordinates. Several criteria can be chosen during this step, including attempting to align nodes, or create straight *long edges*.

Unfortunately, this algorithm also comes with its drawbacks: both in terms of performance and minimizing the amount of edge crossings.

In 2005, Eiglesperger *et al.* [6] provided an improved version of Sugiyama's algorithm, bringing the worst case time complexity from $\mathcal{O}(|V| * |E| * \log(|E|))$ to $\mathcal{O}((|V| + |E|) * \log(|E|))$. This was achieved by reducing the amount of *dummy nodes* created and by allowing edges to span multiple layers. Despite the significant speed-up provided by this algorithm, to the best of our knowledge it is not used in any CFG layout program.

### B. CFG Layout Considerations

As discussed by Devkota *et al.* [5], general graph drawing layouts do not properly represent CFGs. Moreover, they argue that domain experts don't allocate time to improve CFG visualization and instead "spend hours following labyrinthine lines" [5]. To resolve these issues, Devkota *et al.* experimented with many aesthetic changes, such as collapsing functions, collapsing loops, coloring back edges and coloring loops. Instead of starting from scratch, Devkota *et al.* base themselves on Dagre or `dot` [3]. They also never compare their work with existing binary analysis visualization tools.

### C. Two Terminal Layouts

After having written a first version of our tool, we came across an approach very similar to our own by Andrey Mikhailov *et al.* [9]. Their approach is based on separating a CFG into *Two Terminal* regions, and then applying templates for known regions. Although the article discusses Single Entry Single Exit (SESE) regions (used by our approach and detailed in the next section), they do not seem to use them nor do they explain the advantages of *Two Terminal* regions. Last but not

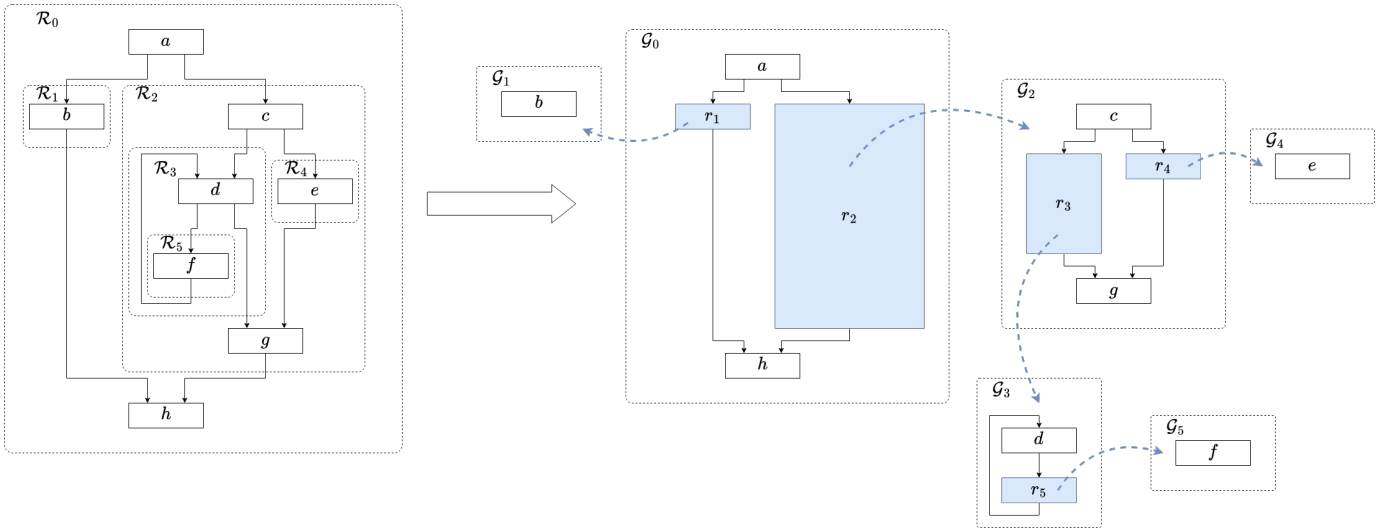



Fig. 4: Splitting a graph into its SESE regions

least, we were unable to find their implementation nor any sort of evaluation on real world binaries.

## IV. Contribution

We decided to split the layout problem into multiple smaller layout problems using Single Entry Single Exit (SESE) regions.

### A. Single Entry Single Exit regions

SESE regions were described by Johnson *et al.* along with an algorithm to identify these regions in linear time (with regard to the amount of edges in the graph) [8].

We recall here a few notions before defining SESE regions. In a CFG $\mathcal{G}$, a node $a$ is said to *dominate* another node $b$, if every path starting from the entry node of $\mathcal{G}$ and going to $b$ must pass through $a$. Said differently, if an execution reaches $b$, then it first went through $a$ – and possibly through other nodes in-between $a$ and $b$. Similarly, a node $b$ postdominates a node $a$ if $b$ dominates $a$ in the dual CFG $\mathcal{G}'$ obtained by reversing edges direction in $\mathcal{G}$.

This notion can be extended to edges: an edge $e$ *dominates* an edge $f$ in $\mathcal{G}$ if every path – starting from the entry point – taking $f$ must take $e$ first.

An SESE region of a graph is a region with a single edge entering the region and a single edge exiting this region. Formally, Johnson *et al.* define it as an ordered edge pair $(a, b)$ of distinct control flow edges $a$ and $b$ such that:

- $a$ dominates $b$.
- $b$ postdominates $a$.
- Every cycle containing $a$ also contains $b$ and vice versa.

If $(a, b)$ and $(b, c)$ are SESE regions, then $(a, c)$ is also an SESE region. Instead of working with all SESE regions, we only consider *canonical* SESE regions: these are the smallest regions for which an edge is an entry or exit edge [8].

SESE regions are interesting as many standard control flow patterns are contained within an SESE region: if statements, loops etc. In general, each C construct that does not contain `gotos` can be mapped to an SESE region [2]. For this reason, SESE regions can be used both in program analysis – for example to convert a program to *Static Single Assignment form* [8] – and in decompilation – for example in SAILR's region identification [2].

### B. Our method: divide-and-conquer based on SESE regions

As noted by Johnson *et al.*, SESE regions can be used as a base for a *divide-and-conquer* approach, since calculating them can be done in linear time. We propose to apply this *divide-and-conquer* approach to graph drawing.

We start by identifying canonical SESE regions in the graph using the algorithm provided by Johnson *et al.*. We provide in appendix A an annotated version of Johnson *et al.*'s algorithm, which we hope can aid in its comprehension.

Then, for each canonical SESE region $\mathcal{R}_i$ in our graph $\mathcal{G}$, we create a new graph $\mathcal{G}_i$. This graph $\mathcal{G}_i$ contains nodes for each SESE region $\mathcal{R}_j$ in $\mathcal{R}_i$ as well as the nodes that do not belong to a sub-region.

Figure 4 shows the rough outline of this step. In the figure, we start with a graph containing 6 SESE regions, we will therefore create 6 graphs. The first one, $\mathcal{G}_0$ contains the nodes $a$ and $h$ that are in the SESE region $\mathcal{R}_0$, as well as two new nodes $r_1$ and $r_2$ which represent respectively regions $\mathcal{R}_1$ and $\mathcal{R}_2$.

Using Sugiyama's layered graph drawing, we can now lay out each of these graphs. We start with the smallest region graphs – with $\mathcal{G}_j < \mathcal{G}_i$ if $\mathcal{R}_j$ is an SESE region of $\mathcal{R}_i$. This way, when we get to $\mathcal{R}_i$, we know the dimension for each region node $r_j$, and can lay $\mathcal{G}_{\mathcal{R}_i}$ out accordingly. For example, we can only know the width of $r_2$ after having laid out $\mathcal{G}_2$.

Once each graph has been laid out, we can draw each region graph in its corresponding region node. Finally, we can reconnect the edges to and from each region's entry and exit node. We have provided the source code for our

crude implementation on a GitHub repository: https://github.com/triskellib/triskel. The implementation can get tedious, notably when connecting edges between SESE regions. Our C++ implementation, which includes the Sugiyama, and SESE regions identification algorithms, is around 5k lines of code.

### C. Advantages of using Single Entry Single Exit regions

Laying out the CFG using SESE regions guarantees that these regions are properly highlighted. We therefore have a guarantee that the layout will contain the patterns we are looking for in CFGs. Another advantage of using SESE regions is that, by splitting up the graph layout problem into multiple smaller problems, we improve the quality of the layout.

### D. Limitations of using SESE regions

In real world binaries, not all regions of interest are SESE regions. This can be due to optimizations or use of `gotos`. For example the CFG in fig. 5 does not contain any large SESE regions.

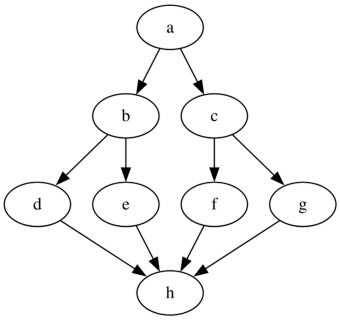
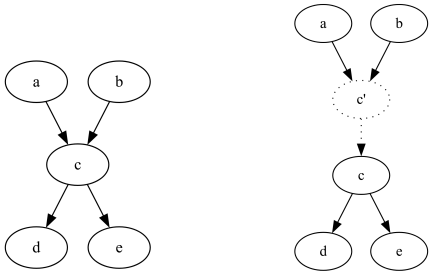



Fig. 5: No large SESE region is identified

*1) Ghost nodes:* We are currently working on rules to add *ghost nodes* to help with SESE regions detection. So far, we have determined that we need to add a *ghost node* to nodes with multiple predecessors and successors, as in fig. 6.




(a) Before *ghost node* (b) With a *ghost node*

Fig. 6: Process of adding a *ghost node*

These simple modifications allowed us to identify 5% more SESE regions. However, it introduced a lot of bugs in our implementation, so *ghost nodes* were not used during the evaluation phase.

It is important to keep the optimization passes linear to avoid jeopardizing *divide-and-conquer* advantages.

*2) Other patterns:* Unfortunately, some patterns are not identifiable even with *ghost nodes*: early `returns`, multiple exit loops and the use of `gotos` can create *patterns* that do not fit neatly within SESE regions. This does not invalidate the current algorithm, however its efficiency is limited in these situations. Fortunately, identifying `gotos` and early returns is a large part of *structural decompilation*. We believe that this method can be significantly improved by applying strategies used in decompilation to graph visualization.

## V. Modifications to existing algorithms

During our research, we also experimented with small modifications to the existing Sugiyama layering algorithm, which yielded major improvements. These might not be novel, as Sugiyama has many implementations and variations, however we have shown that these modifications specifically impact CFG layouts.

### A. Sliding nodes

If a node with a single entry and single exit edge is placed before or after a long edge, we try "sliding" the node along this long edge (creating 2 long edges instead of 1) in order to minimize the height of the graph. To do this:

- We start by identifying each potential *sliding node* – a node with a single entry and exit edge, one of which is a long edge ;
- In decreasing node height order, we try assigning the nodes to each possible layer – between its parent and its child, measuring the resulting graph height ;
- We pick the layer that minimizes the graph height.

Figure 7 illustrates this process and the resulting gain, with a single sliding node.

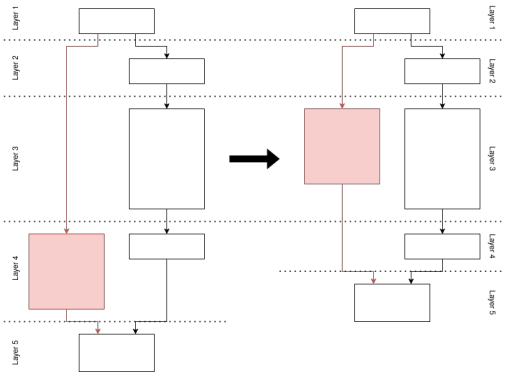



Fig. 7: By sliding the red node, we are able to reduce the graph height.

This can lead to a wider yet shorter graph, while also creating shorter edges which are easier to understand.

In our *LLVM benchmark* for `serde`, we found that sliding nodes could be performed 8.6% times, yielding graphs that were up to 30% shorter, and 3% shorter on average. The width increase was smaller, at 1% on average when sliding was performed.

| | Triskel | IDA | Iato | Binary Ninja | angr | Ghidra [1] | Ghidra [2] |
|---|---|---|---|---|---|---|---|
| 2B90 | **1** | 6 | 8 | 18 | 30 | 5 ◇ | 2 ◇ |
| 3110 | 2 | **1** | 2 | 18 | 23 | 4 ◇ | 4 ◇ |
| 3220 | **0** | 1 | 5 | 15 | 29 | 3 ◇ | **0** |
| 3400 | 6 | **0** | 5 | 9 | $> 95$ | 5 ◇ | 3 ◇ |
| 3540 | **0** | 1 | 1 | 4 | 5 | 5 ◇ | 1 ◇ |
| 3C40 | **4** | **4** | 14 | 35 | 26 | 21 ◇ | 11 |
| 3D60 | **0** | **0** | **0** | 1 | 1 | **0** | **0** |
| 3E20 | **0** | 3 | 3 | 4 | 5 | 3 ◇ | 0 ◇ |
| 5AC0 | 4 | **1** | 6 | 11 | 27 | 11 ◇ | 1 ◇ |
| 5DE0 | 17 | **11** | 16 | 77 | 68 | 20 ◇ | 14 ◇ |

TABLE I: Number of intersections in CFG layouts on selected `md5sum` functions
**bold** items indicate layouts with the fewest crossings, ◇ indicate layouts where an edge intersects a node

[1] Using Ghidra's default layout engine: *Nested Code Layout*.
[2] Using Ghidra's *Hierarchical MinCross Network Simplex* layout.

### B. Increasing node mobility

During node ordering, we found that shuffling the array and favoring permutations slightly reduced the number of intersections. This increased mobility led to 0.06 fewer intersections per function on average, due to most graphs already having 0 intersections. However, if we consider only graphs which had intersections, we have 0.1 fewer intersections.

## VI. Evaluation

As discussed previously, the ideal graph layout is the one which maximizes clarity. However, this is a difficult metric to measure. Instead, we mainly focused on the number of intersections that were found when laying functions out.

We would like to highlight that our method provides some additional *semantic clarity* by construction. Indeed, structures analysts are after, such as `ifs` and `whiles`, will be grouped by design. This is hard to illustrate in the general case, however, fig. 19 contains a *somewhat cherry-picked* example of this advantage.

### A. Binaries

We compared our tool with other layout engines used in binary analysis frameworks, the results are available in section VI-A. This evaluation was done on functions[1] from the `md5sum` Linux utility. These functions were picked to avoid switch statements and maximize the amount of basic blocks.

The metric we picked for the quality of a layout is the number of crossings. As discussed in II-A, this metric is imperfect but is easy to measure. We counted each intersection rather than simply intersection areas. This means fig. 8 contains 44 intersections.

[1] we reference them by their address

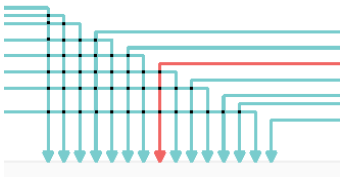


Fig. 8: An extract of a CFG rendered with `angr` management with 44 intersections (black points)

We compared the following tools:

- IDA Free (version `9.0.240925`);
- Iato, the GUI of `radare2` (version `5.9.9`);
- Binary Ninja Free (version `426455-Stable`);
- `angr` Management, the GUI of `angr` (version `9.2.136`);
- Ghidra (version `11.2.1`).

Since these tools aim to display CFGs, intended for human analysts, they do not provide simple APIs to interact directly with the layout. As a result, we had to **manually** count intersections and nodes for each layout, which is why we had to limit our analysis to 10 functions.

Another major limitation of this evaluation is that, although the functions are the same, the graphs are not. In particular, the number of nodes varies significantly depending on the tool, as shown in appendix D. These fluctuations are caused by numerous factors. For Binary Ninja, the graph contains pseudocode rather than the actual assembly listing. In other cases, differences can arise from the underlying binary analysis engine detecting, or not, inlined function and long jumps. These differences might have been even more pronounced had we chosen functions with indirect jumps.

Overall, these results might not be fully representative of each tool's abilities, given that they were obtained on a very small dataset, and the intersections were counted manually. However, our tool seems almost on par with IDA, and is often able to find the best layout out of all the tools in this evaluation.

It is also worth noting that Ghidra's layout engine allows edges to intersect with nodes. We did not count such crossings,

but they do significantly take away from the graph's readability. An example of this is available in appendix C.

For illustration purposes, we have provided each tool's layout of the function at address `3540` in appendix B.

### B. LLVM

The first step to laying a CFG out is retrieving it, usually by disassembling the binary code of the program. As a result, differing layout results for different tools may be due to differing disassemblers, rather than differences in their layout engines. The problems of disassembly and CFG layouts are orthogonal.

In order to compare layout engines fairly, we experimented with displaying the LLVM Intermediate Representation (IR) CFGs.

This offers several advantages:

- Access to the CFG through LLVM's API;
- Access to structures, which are hard to disassemble but important in layouts (e.g switch statements);
- Facilitates the creation of a benchmarking tool.

Furthermore, although the CFG will not be exactly the same as with the machine code, it is very similar and often the LLVM CFG is more complicated.

In order to properly benchmark our tool, we decided to run it on functions from `serde` and `curl`. `serde` is a Rust framework for serialization, its LLVM bytecode is 8.8 MiB. `curl` is a C utility for interacting with networks, its LLVM bytecode is 1.2 MiB.

There needs to be at least 3 blocks for intersections to occur, so we left out functions with 1 or 2 basic blocks. We also left out functions with more than 300 edges, for performance reasons.

This left us with 4,860 functions, which took our layout engine 11 min 10s to process. In these functions, we found 21,038 SESE regions. Impressively, 3,904 functions (80%) were laid out without any intersections.

Figures 9 and 10 illustrate statistics on the number of nodes – aka blocks – per function, and the number of intersections per nodes. In our dataset, as expected, most functions have few blocks and can be laid out without intersections, while the larger functions have more intersections.

## VII. Limitations

The main limitation of our tool, in its current form, is the performance of our Sugiyama implementation. As shown in our results, it works well as a proof-of-concept on small functions, but can take upwards of a minute for functions with more than 300 edges. This is most likely due to the naive way we count intersections during the "node ordering" phase.

Another limitation is that our Sugiyama implementation currently does not support looping edges – edges which connect a node to itself.

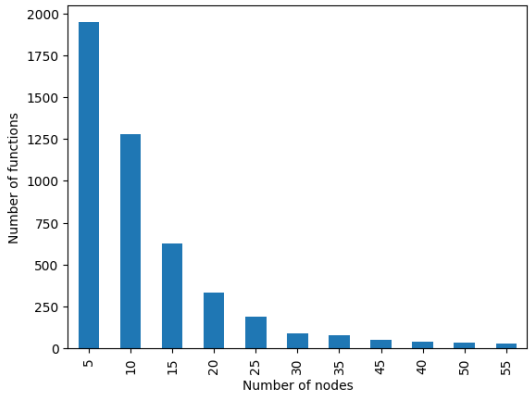



Fig. 9: Distribution of basic blocks across functions

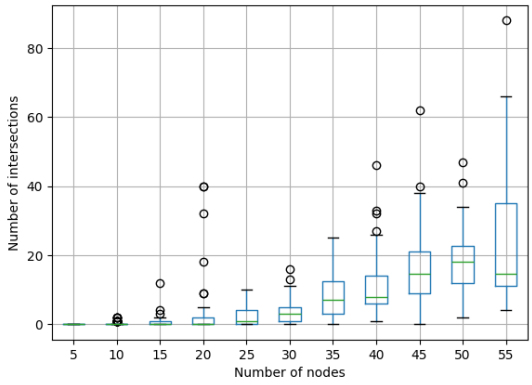



Fig. 10: Box plot of intersections across basic block count

## VIII. Future Work

We hope to fix our performance issues by implementing Eiglsperger's improved version [6] of Sugiyama's algorithm.

We also intend to continue our work on adding *ghost nodes*, in order to detect more SESE regions.

## IX. Conclusion

In this paper, we explained and evaluated a *divide-and-conquer* based approach for CFG layout using SESE regions. In our comparison with various binary analysis tools, we found our tool to be comparable to IDA. We then offered a different evaluation strategy, using LLVM IR CFGs. The C++ PoC of our algorithm is open source, and we hope can be used as a base for CFG layouts.


## Acknowledgment

The authors would like to thank the teams behind `rev.ng` and Binary Ninja for answering our inquiries and for providing valuable advice. This work was funded by "Direction Générale de l'Armement" through CREACH LABS under the project "Mermaid" and by the "France 2030" government investment plan managed by the French National Research Agency, under the reference "ANR-22-PECY-0005".

## REFERENCES


[1] Vector 35. Binary Ninja. https://binary.ninja/.

[2] Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. Ahoy SAILR! there is no need to DREAM of c: A Compiler-Aware structuring algorithm for binary decompilation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 361–378, Philadelphia, PA, Aug. 2024. USENIX Association.

[3] Sabin Devkota and Katherine E. Isaacs. Cfgexplorer: Designing a visual control flow analytics system around basic program analysis operations. *Computer Graphics Forum*, 37(3):453–464, 2018.

[4] Sabin Devkota, Matthew P. LeGendre, Adam J. Kunen, Pascal Aschwanden, and Kate Isaacs. Cfgconf: Supporting high level requirements for visualizing control flow graphs. *ArXiv*, abs/2108.03047, 2021.

[5] Sabin Devkota, Matthew P. LeGendre, Adam J. Kunen, Pascal Aschwanden, and Kate Isaacs. Domain-centered support for layout, tasks, and specification for control flow graph visualization. *2022 Working Conference on Software Visualization (VISSOFT)*, pages 40–50, 2021.

[6] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An efficient implementation of sugiyama's algorithm for layered graph drawing. *Journal of Graph Algorithms and Applications*, 9(3):305–325, Jan. 2005.

[7] HexRays. IDA. https://hex-rays.com/ida-pro.

[8] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: computing control regions in linear time. *SIGPLAN Not.*, 29(6):171–185, Jun. 1994.

[9] Andrey Mikhailov, Aleksey Hmelnov, Evgeny Cherkashin, and Igor Bychkov. Control flow graph visualization in compiled software engineering. In *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1313–1317, 2016.

[10] NSA. Ghidra. https://ghidra-sre.org/.

[11] radare. Iato. https://rada.re/n/iaito.html.

[12] rev.ng Labs. rev.ng. https://rev.ng/.

[13] SoftSec Lab, Korea Advanced Institute of Science & Technology. B2R2. https://github.com/B2R2-org/B2R2.

[14] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

## APPENDIX A
## ALGORITHMS

```
1: procedure CYCLEEQUIV(G)
2:     for each node n in reverse depth-first order do
3:         /* compute n.hi */
4:         hi_0 := min{t.dfsnum | (n,t) is a backedge};
5:         hi_1 := min{c.hi | c is a child of n};                ▷ child refers to DFS child in the undirected graph
6:         n_hi := min{hi_0, hi_1};
7:         hichild := any child c of n having c.hi = hi_1;                ▷ hichild is a node not a set
8:         hi_2 := min{c.hi | c is a child of n different than hichild};   ▷ since hichild is a node, we can have hi_1 = hi_2
9:
10:        /* compute bracketlist */
11:        n.blist := create();
12:        for each child c of n do                              ▷ child refers to DFS child in the undirected graph
13:            n.blist := concat(c.blist, n.blist);
14:        end for
15:        for each capping backedge d from a descendent of n to n do   ▷ descendent refers to DFS descendent in the
   undirected graph
16:            delete(n.blist, d);
17:        end for
18:        for each backedge b from a descendent of n to n do   ▷ descendent refers to DFS descendent in the undirected
   graph
19:            delete(n.blist, b);
20:            if b.class undefined then
21:                b.class := new_class();
22:            end if
23:        end for
24:        for each backedge e from n to an ancestor of n do   ▷ ancestor refers to DFS ancestor in the undirected graph
25:            push(n.blist, e);
26:        end for
27:        if hi_2 < hi_0 then
28:            /* create capping backedge */
29:            d := (n, node[hi_2]);
30:            push(n.blist, d);
31:        end if
32:
33:        /* determine class for edge from parent(n) to n */
34:        if n is not the root of the dfs tree then
35:            let e be the tree edge from parent(n) to (n)      ▷ parent refers to DFS parent in the undirected graph
36:            b := top(n.blist)
37:            if b.recentSize ≠ size(n.blist) then
38:                b.recentSize := size(n.blist)
39:                b.recentClass := new_class()
40:            end if
41:            e.class = b.recentClass
42:
43:            /* check for e, b equivalence */
44:            if b.recentSize = 1 then
45:                b.class := e.class;
46:            end if
47:        end if
48:    end for
49: end procedure
```

Fig. 11: An annotated version of the cycle equivalence algorithm from Johnson *et.al.* [8]

## APPENDIX B
## GRAPHS

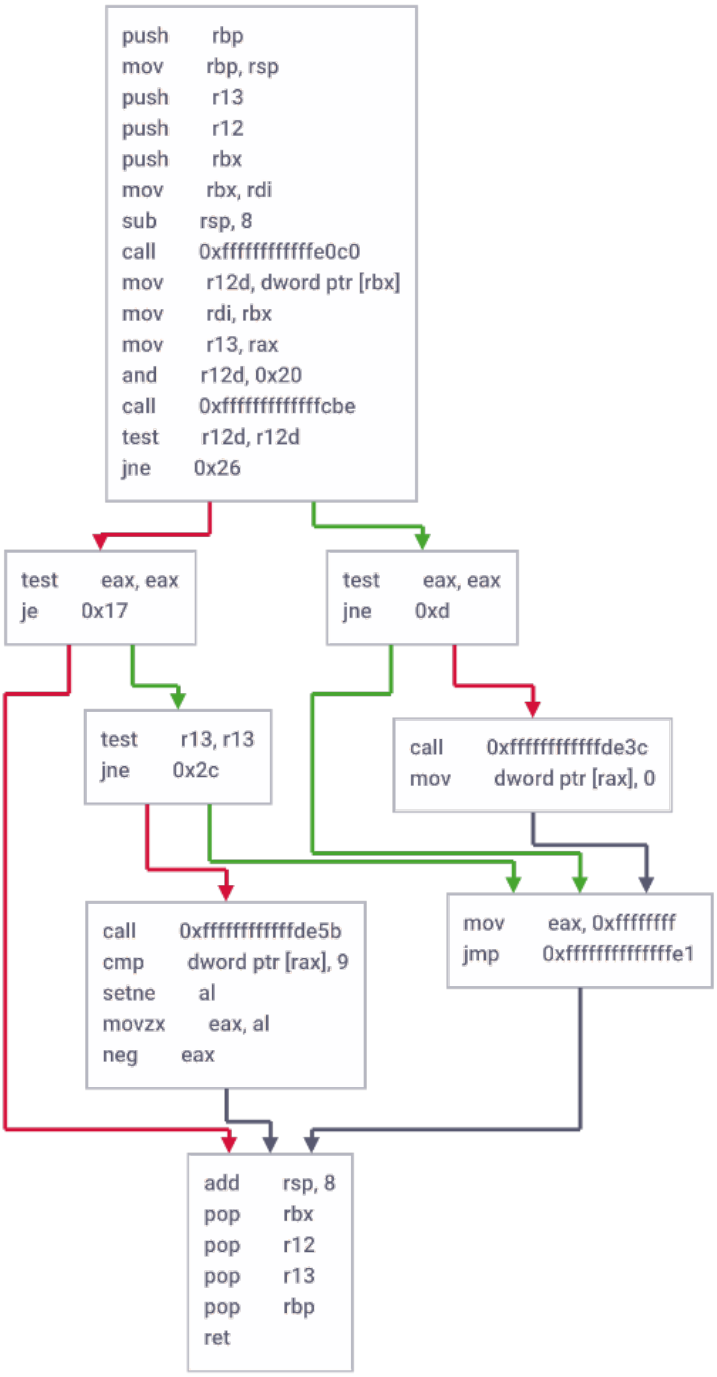



Fig. 12: `md5sum`'s function at `3540`, laid out by our tool

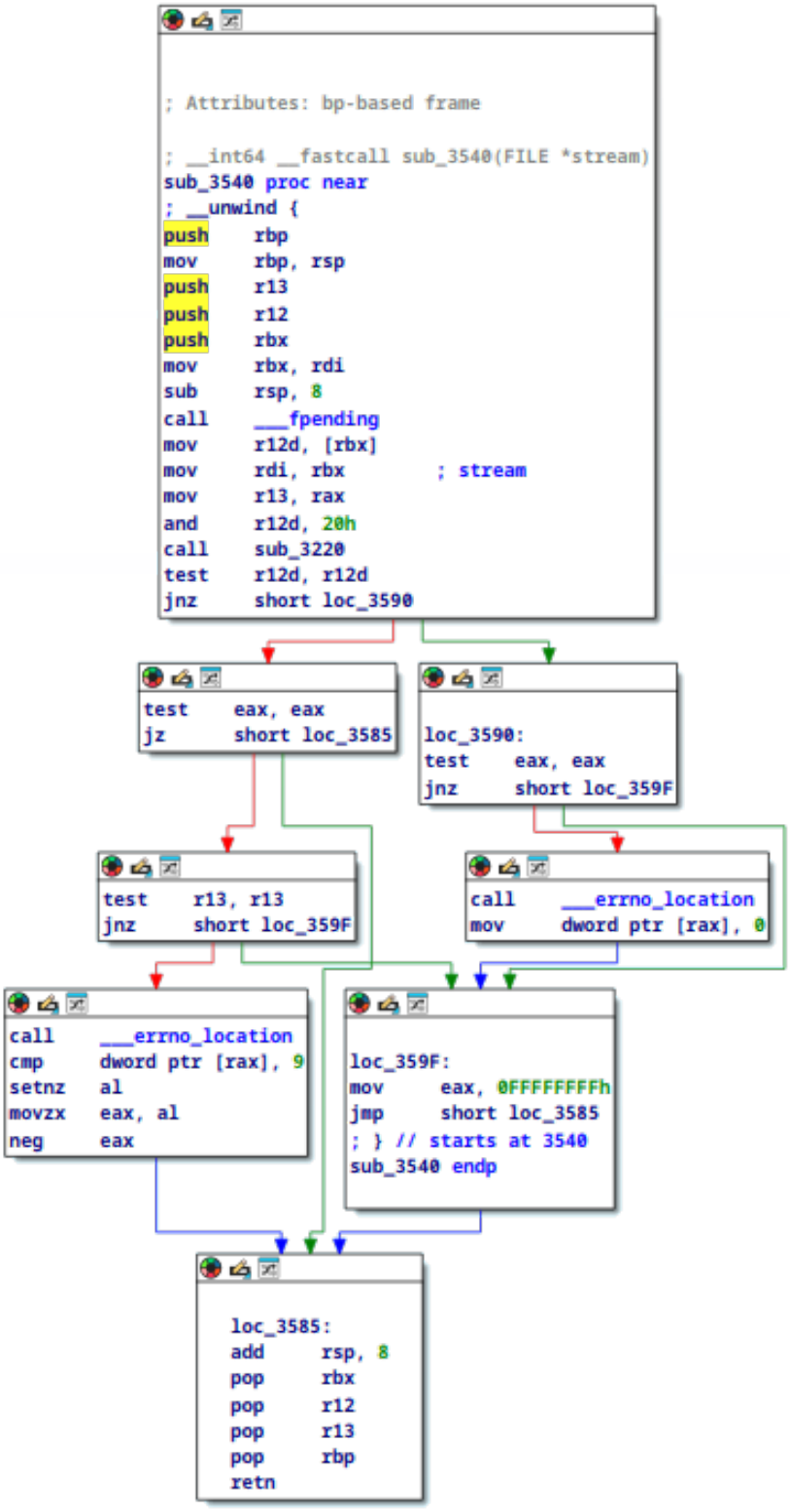



Fig. 13: `md5sum`'s function at `3540`, laid out by IDA

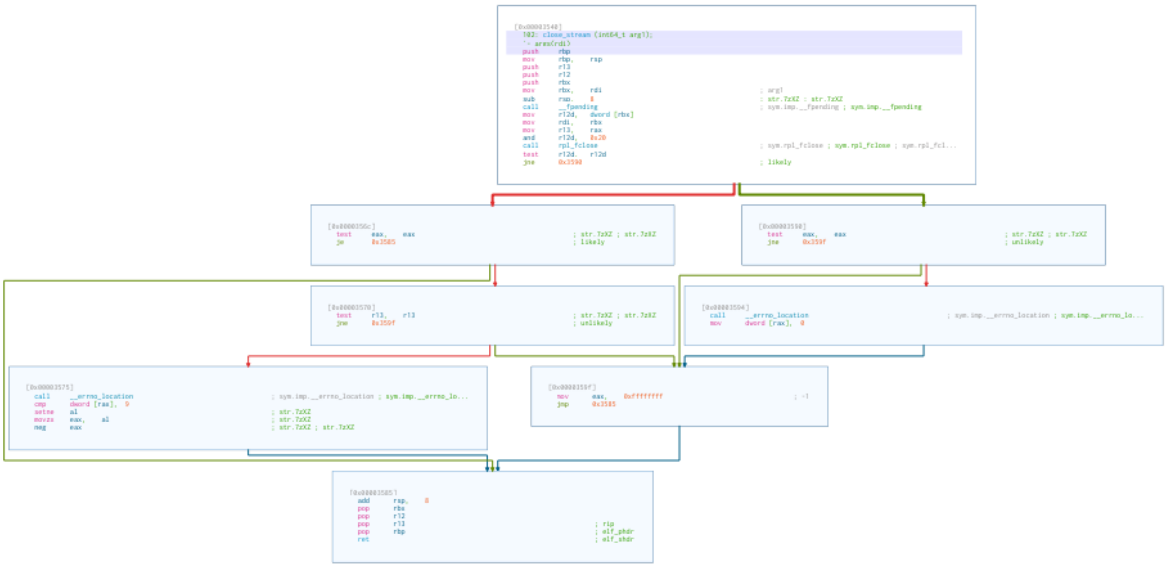


Fig. 14: `md5sum`'s function at `3540`, laid out by Iato

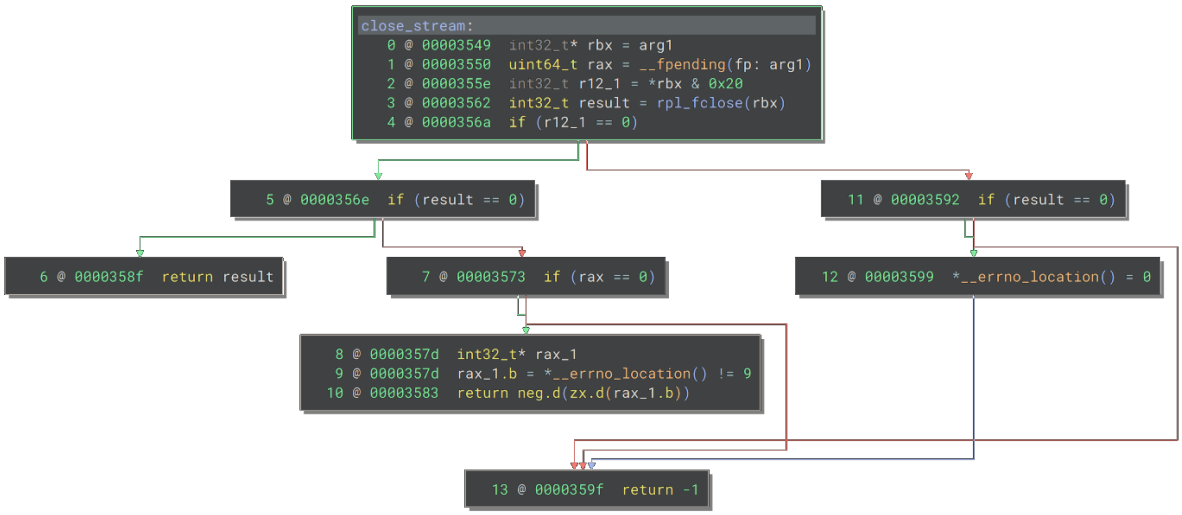



Fig. 15: `md5sum`'s function at `3540`, laid out by Binary Ninja

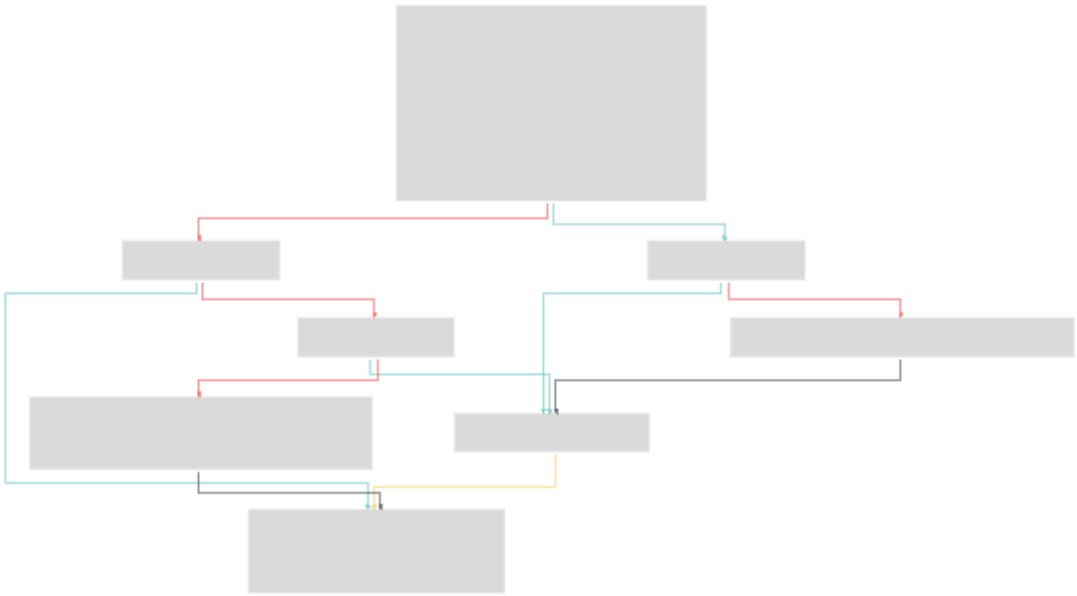


Fig. 16: `md5sum`'s function at `3540`, laid out by `angr`[2]

[2]When zooming out, `angr` management removes the instructions from the blocks.

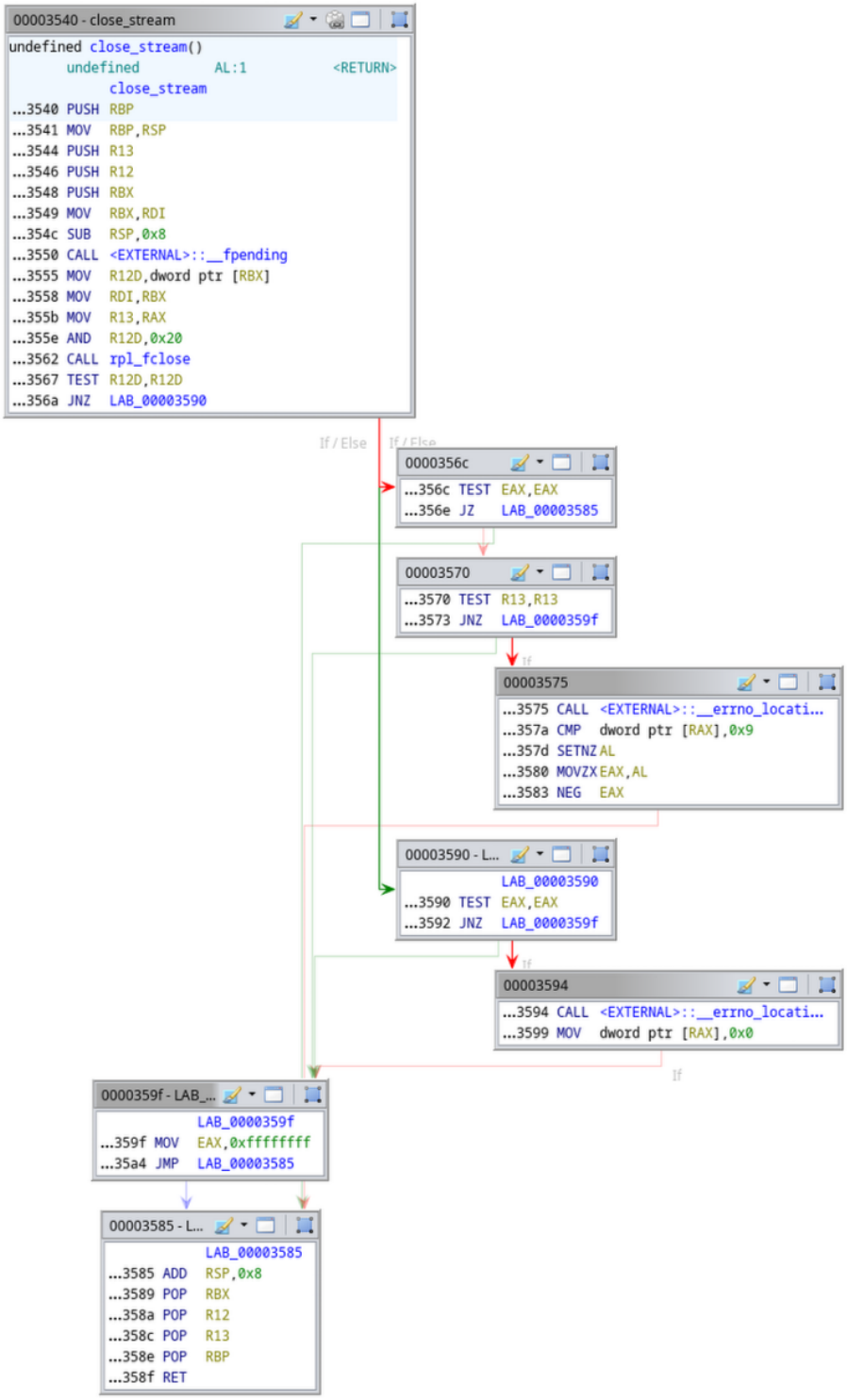



Fig. 17: `md5sum`'s function at `3540`, laid out by Ghidra's default layout engine: *Nested Code Layout*

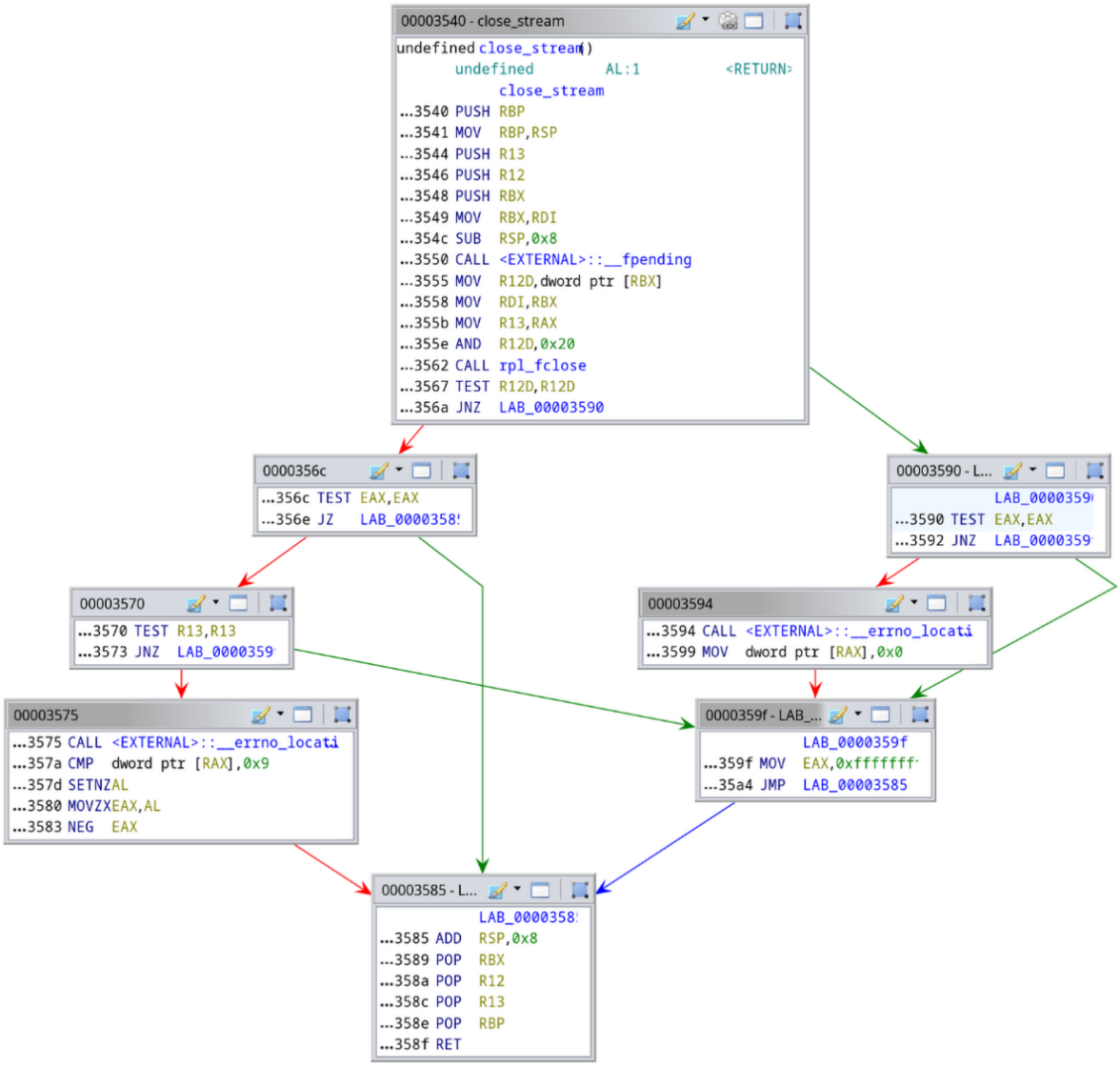



Fig. 18: `md5sum`'s function at `3540`, laid out by Ghidra's *Hierarchical MinCross Network Simplex* layout engine

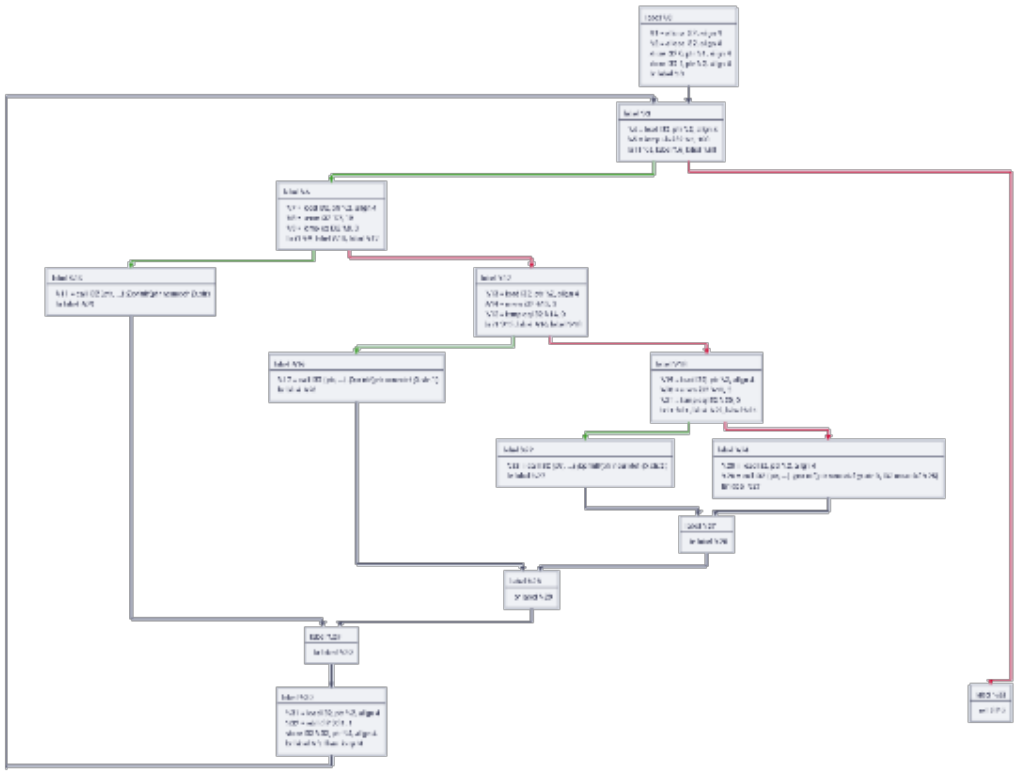


Fig. 19: Our layout of a *fizzbuzz* program compiled with no optimization (`-O0`) to clearly highlight the SESE regions.

## APPENDIX C
## EDGE CROSSING A NODE

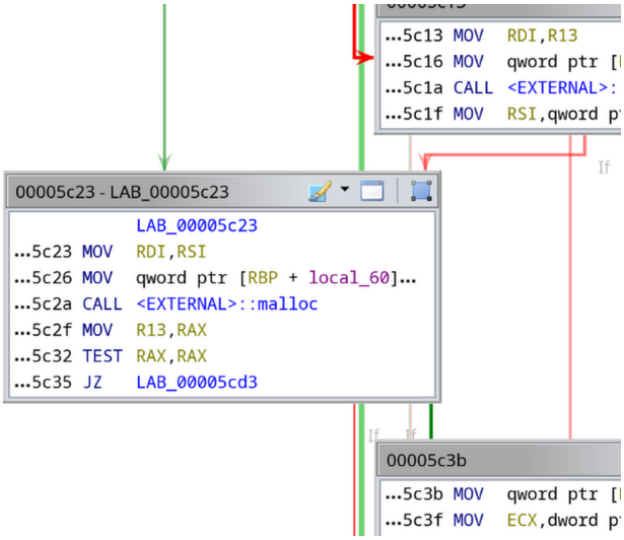



Fig. 20: An example of an edge (the green one in the middle) crossing a node in Ghidra's layout.

## APPENDIX D
## VARYING BASIC BLOCK COUNT

| | Our tool | IDA | Iato | Binary Ninja | `angr` | Ghidra |
|---|---|---|---|---|---|---|
| `2B90` | 22 | 22 | 21 | 28 | 22 | 22 |
| `3110` | 16 | 15 | 15 | 18 | 15 | 16 |
| `3220` | 18 | 17 | 17 | 15 | 17 | 17 |
| `3400` | 19 | 19 | 19 | 14 | 19 | 19 |
| `3540` | 8 | 8 | 8 | 8 | 8 | 8 |
| `3C40` | 23 | 22 | 22 | 37 | 23 | 22 |
| `3D60` | 14 | 5 | 5 | 5 | 5 | 5 |
| `3E20` | 10 | 10 | 10 | 10 | 10 | 10 |
| `5AC0` | 19 | 19 | 19 | 19 | 19 | 19 |
| `5DE0` | 38 | 38 | 38 | 52 | 38 | 38 |

TABLE II: Number of nodes in CFG layouts on selected `md5sum` functions