SoK: No Goto, No Cry? The Fairy Tale of Flawless Control-Flow Structuring

Eva-Maria C. Behner
Fraunhofer FKIE
Bonn, Germany
eva-maria.behner@fkie.fraunhofer.de

Steffen Enders
Fraunhofer FKIE
Bonn, Germany
steffen.enders@fkie.fraunhofer.de

Elmar Padilla
Fraunhofer FKIE
Bonn, Germany
elmar.padilla@fkie.fraunhofer.de

***Abstract*—Decompilers play a crucial role in the detailed analysis of malware or firmware, particularly because control-flow structuring allows the recovery of high-level code that is more readable to human analysts. Despite the ongoing debate over their usage of gotos to work around constraints during control-flow structuring, pattern-matching approaches remain prevalent among both commercial and open-source decompilers. With the emergence of pattern-independent restructuring techniques, various attempts have been made to overcome readability limitations, especially concerning the use of gotos. However, despite these advances, recent approaches often fail to thoroughly address several inherent challenges of control-flow structuring, thereby affecting output quality or practicality.**

In this paper, we systematize the intrinsic challenges of control-flow structuring that every approach must address. In addition, we review existing methods, comparing them, while highlighting both their advantages and limitations with respect to these challenges. Specifically, we emphasize the practicability issues of current pattern-independent restructuring techniques and discuss whether and how future methods might overcome them. Finally, we explore the theoretical potential to mitigate some of these challenges by suggesting methodology ideas for various aspects of control-flow structuring. Overall, this paper enables other researchers to make informed decisions when developing or enhancing control-flow structuring methods, thereby preventing negative side-effects arising from the interdependence of challenges.

***Index Terms*—decompilation, control-flow recovery, control-flow structuring, reverse engineering, static analysis**

1. Introduction

In numerous instances involving binary analysis, the source code is inaccessible, particularly in the context of malware analysis [58]. To fully comprehend the binary and achieve the desired analysis objectives, analysts typically employ a decompiler to generate a high-level representation that accurately reflects the program semantics, ensuring an accurate reflection of its intended operations and behaviors [49]. Although there are other use cases, including recompilability [66], the predominant focus of this paper is on decompilation from the perspective of manual analysis, emphasizing the importance of *clarity* and *readability* [8], [25]. In light of the significance of this analysis step, there is a plethora of decompilers and decompilation approaches that focus on facilitating manual analysis [4], [7], [12], [27], [38], [73].

While these decompilers often use similar data-flow analyses, there are notable differences and limitations in the crucial step of control-flow structuring [23]. This step is essential for converting control-flow graphs to high-level structures in order to gain clarity and organization [63]. By converting these graphs into clear, high-level code with essential constructs such as loops and if-conditions, the readability is improved and the efficiency of analysts is notably increased [67]. Usually, current methods for reconstructing high-level programming structures fall into two categories: *Pattern-matching* and *pattern-independent*, each having its own benefits and justifications. Unfortunately, achieving a perfect solution is merely possible, consequently necessitating certain trade-offs [23]. Although each design choice can be justified within a specific context, they result in inherent limitations that need to be thoroughly discussed.

Pattern-matching approaches are commonly used in both commercial and open-source decompilers [38], [51], as they provide a reliable and practical way to produce decompiled output. This is achieved by maintaining a collection of predefined patterns, with each pattern matching a subgraph, that can be directly interpreted as C-structures. However, due to the magnitude of different compiler versions, settings, and optimizations, compiling a comprehensive pattern-set is an extremely challenging task, even for state-of-the-art decompilers. As a result, for subgraphs that do not fit any existing pattern, these techniques can introduce *gotos*, even when a simple C-structure is available. Unfortunately, this can result in an output that is much less readable [22].

With the introduction of DREAM, pattern-independent restructuring approaches have emerged as potential solutions to the aforementioned problems [27], [35], [73]. Although these methods do not depend on predefined patterns, they may suffer from runtime issues when applied to large functions or deliver suboptimal results. Nevertheless, it is important to acknowledge that, due to the inherently academic orientation of these methods, practicability is often not their primary goal. In fact, most of these approaches are able to achieve their goal of producing a goto-free decompiler output. However, in order to achieve this objective, they frequently employ techniques such as introducing structural variables, which can unintentionally increase cognitive complexity [9]. In conclusion, depending on the specific case, the output might be more readable than that generated by pattern-matching methods, but understanding the constructs introduced to avoid gotos can be even more time-consuming.

© 2025, Eva-Maria C. Behner. Under license to IEEE.
DOI 10.1109/EuroSP63326.2025.00032

The shortcomings inherent in both types of approaches have been addressed by various researchers proposing a variety of different modifications and adaptations [4], [27], [35], [71]. Specifically, most of these strategies pinpoint a particular flaw in current methods and aim to address it accordingly. However, although one particular aspect is often successfully improved, the accompanying limitations and issues are not always adequately examined or discussed. For example, both approaches, pattern-matching [38] and pattern-independent [21], [35], may duplicate parts of the code to reduce the number of gotos. Nevertheless, there has been no comprehensive discussion on how this enhances readability and when it complicates analysis; because duplicating code can clearly reduce the amount of gotos but also inevitably increases code size. Specifically, in cases where complex structures or long code segments are duplicated, such a strategy becomes questionable.

The problem with contemporary approaches seems to be their heavy reliance on established ideas while still constraining themselves by only considering a subset of available approaches. Whether these limitations are intentional or accidental is often unclear, as they are usually not discussed. In particular, approaches frequently fail to adequately address several challenges when recovering high-level structures from the control-flow graph. Although some methods successfully address specific aspects, their authors frequently overlook other problems or fail to consider potential downsides, focusing only on the advantages. This, combined with the fact that many authors do not clearly state the limitations they (un)knowingly accepted, makes it difficult to compare or pursue new approaches. In order to address this problem effectively, we argue for the necessity of compiling a comprehensive list of known challenges associated with control-flow structuring.

In this paper, we thoroughly investigate existing research on control-flow structuring to allow for better categorization and understanding of existing approaches. Furthermore, this enables for a more objective and accurate comparison of existing approaches. In addition, this facilitates conscious decisions during the development or improvement of methods for control-flow structuring. **Overall, we make the following three contributions:**

I: We identify which decisions must be made during control-flow structuring and outline the four main challenges that arise (see Figure 1). Additionally, we highlight that they are interdependent and inherently unsolvable, implying that each method must balance trade-offs between output quality and practicability.

II: Building on the previously identified challenges, we examine current approaches in order to allow for a better comparison. This involves highlighting both the advantages and limitations of the various methods. Here, we specifically highlight undiscussed limitations of pattern-independent approaches.

III: We discuss potential solutions to these challenges and limitations from a theoretical point of view. Although this theoretical proposal still needs to resolve certain decisions, such as determining when to duplicate, it can be customized for various requirements and forms a robust base for future research.

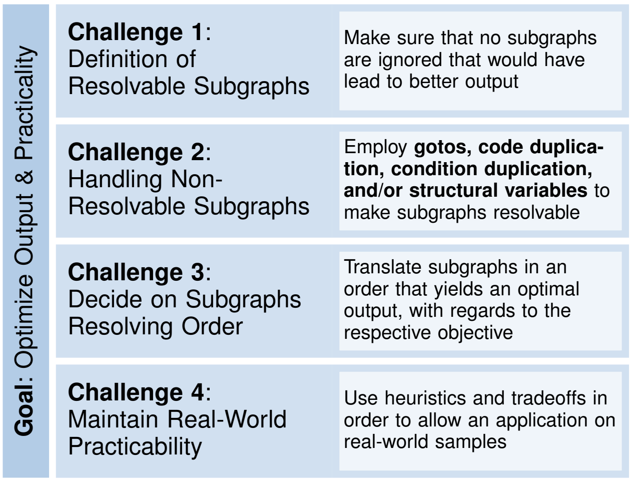

Goal: Optimize Output & Practicality
Challenge 1: Definition of Resolvable Subgraphs
Make sure that no subgraphs are ignored that would have lead to better output
Challenge 2: Handling Non-Resolvable Subgraphs
Employ gotos, code duplication, condition duplication, and/or structural variables to make subgraphs resolvable
Challenge 3: Decide on Subgraphs Resolving Order
Translate subgraphs in an order that yields an optimal output, with regards to the respective objective
Challenge 4: Maintain Real-World Practicability
Use heuristics and tradeoffs in order to allow an application on real-world samples

Figure 1: The four challenges in control-flow structuring.

2. Related Work

Control-flow structuring, or high-level control-flow recovery, remains a pivotal aspect of the decompilation process and thus has been a significant research area for many years. While translating high-level control-flow constructs into a control-flow graph is comparatively straightforward, performing the reverse, *control-flow structuring*, is considerably more challenging. Specifically, the lossy compilation process, along with various compiler optimizations, poses major obstacles in identifying these high-level control-flow constructs. Generally, methods work by iteratively converting a subgraph of the control-flow graph (in short *CFG*) into C-structures and then contracting the subgraph into a single node. This process continues until the complete CFG is *translated*, i.e., contracted into a single node. In the following, we will provide an overview of relevant existing research on control-flow structuring.

A major portion of the techniques relevant to modern approaches originate in the research area around data-flow analysis and compilation [1], [18]–[20], [41], [59], [62]. Specifically, *interval analysis* was initially introduced for compilation purposes [18], [19] and later expanded by *structural analysis* [62], and similar high-level data-flow analysis techniques [59]. Essentially, structural analysis, or *pattern-matching* approaches, identifies specific subgraphs, known as *patterns*, within the CFG that can be transformed into code constructs such as if statements or loops. Building on this concept, Cifuentes [14]–[17] established the foundation for modern decompilers that use pattern-matching to translate subraphs of the CFG into high-level control-flow structures [4], [7], [38], [51]. Notably, the structure of the CFG can sometimes prevent the existence of subgraphs that match any pattern, necessitating the use of alternatives like gotos to proceed with the recovery process of high-level structures.

Since then, several decompilers, such as Ghidra [51] and Hex-Rays [34], have been developed on the basis of this concept, with many efforts to further enhance them. For example, graph transformations have been utilized to improve the identification of subgraphs that match specific patterns [7], [38]. Moreover, various methods have been suggested to avoid gotos or reduce the complexity of the code. One such method employed by Hex-Rays involves

	Region Identification	Utilized Modifications	Particularities & Main Limitation
Binary Ninja [65]	**Pattern-independent** with single entry/succ. regions	No structural variables	Duplicating inside regions but not to get new ones; ignores single exit regions potentially leading to missing regions and poor output.
dewolf [27]	**Pattern-independent** with SISO $\subseteq$ dominance region	Using no gotos and no code duplication	Always structuring the whole smallest ($n > 2$) SISO region, which can lead to large regions & conditions causing long computations.
DREAM [73]	**Pattern-independent** with dominance $\wedge$ SISO regions	Using no gotos and no code duplication	Missing SISO regions by only considering the dominance region, which results in large regions & conditions causing long computations.
Ghidra [51]	**Pattern-matching**	Using solely gotos	Seemingly strict patterns and issues with loop-structuring resulting in potentially missing regions that may cause poor output.
Hex-Rays [38]	**Pattern-matching**	Using gotos and *some* code duplication	Despite the seemingly larger pattern set & improved loop-structuring, still potentially missing regions that may cause poor output.
Phoenix [7]	**Pattern-matching**	Using soley gotos	Although extending loops to reduce gotos, the limited amount of patterns potentially leads to missing regions that may cause poor output.
radare2 [56]	**Pattern-matching**	Using soley gotos	Restricted pattern-set results in missing high-level structures and, combined with nearly no data-flow optimizations, in assembly-like output.
RetDec [26]	**Pattern-matching**	Using soley gotos	Structuring seems not to be the main focus; utilizing only very few patterns & potentially missing regions that may cause poor output.
rev.ng [35]	**Pattern-independent** with extensive preprocessing	Using no gotos	Duplicating code to manage region-size & runtime, potentially resulting in output containing a lot of duplicated code.
SAILR [4]	**Pattern-matching**	Using gotos and some code (de)duplication	Optimizing source code proximity by reverting compiler optimizations; still potentially missing regions that may cause poor output.

TABLE 1: Overview about decompilers, each with an assessment of region identification, utilized modifications, and particularities and resulting limitations (based on corresponding papers, available documentation and our observations).

the duplication of return-statements. Engel et al. [29] also successfully reduced the number of gotos by extending structural analysis to account for C-specific control statements like *break*, *continue*, and *return*. Additionally, the Phoenix decompiler [7] aimed to reduce gotos by proposing an *iterative refinement* approach: initially handling all nodes matching a pattern before executing a *last resort refinement* that removes an edge from the graph while preserving the control-flow using gotos. In this process, they prioritize edges where the source does not dominate the sink, nor does the sink dominate its source, over removing an arbitrary edge. Recently, the authors of SAILR proposed only removing gotos introduced by compiler optimizations, essentially performing de-optimizations to enhance the decompiler output [4].

As the first *pattern-independent* approach, DREAM proposed an alternative to structural analysis that operates without predefined patterns and generates decompiler output without gotos [73]. Unlike methods that depend on predefined patterns, their approach involves using a more flexible definition of subgraphs that can be translated into high-level C-structures. Essentially, if a subgraph has a single entry and a single successor, it can be translated into a C-structure. To ensure that it is *semantics-preserving* and to avoid gotos, DREAM duplicates certain conditions and introduces new variables and conditions accordingly. Since the initial DREAM proposal, several variations of pattern-independent restructuring have emerged [27], [35]. In particular, rev.ng [35] presented another pattern-independent and goto-free restructuring approach by allowing duplication of code and introducing structural variables. These methods help to achieve a well-formed CFG, where each node and its successors in the dominator-tree can be transformed into a natural C-structure. Recently, Binary Ninja [68] significantly enhanced their high-level control-flow recovery technique by integrating a combination of rev.ng [35] and DREAM [73]. Given a single-entry, single-successor region, they translate it into a graph where each node has at most one predecessor, or into a graph containing a subgraph that satisfies this property, using code and condition duplication alongside the inclusion of gotos. Such graphs can be easily translated, provided that the innermost possible region is translated first. In summary, not all pattern-independent methods avoid gotos; in fact, some approaches do utilize gotos additionally to structural variables, code or condition duplications [68].

During the research for this paper, we have reviewed numerous additional decompilers and approaches, including, but not limited to, [5], [6], [32], [50], [61], [66]. However, instead of detailing each of them, we will generally focus on providing meaningful examples and illustrations to demonstrate our key points, primarily because many share similar concepts or emphasize alternative readability features aside from control-flow structuring. Table 1 provides a summary of the decompilers with existing and working implementations that we analyzed more thoroughly and will cover in our evaluation in Section 4.5.

Of course, there are several other areas of decompilation that do not directly relate to control-flow structuring and, as such, are not elaborated upon in detail in this section. For example, multiple studies have been published on the assessment of decompiler accuracy, efficiency, and quality [11], [30], [48], [76]. Additionally, there have been various approaches to improve decompilation apart from control-flow structuring. In particular, several approaches have been suggested for the recovery of types [44], [46], [47], [64], [74], [75]. Correspondingly, numerous studies have focused on meaningfully naming recovered variables and/or functions to enhance the clarity of the decompiler output [24], [36], [40], [43], [52], [70].

Finally, there have been various approaches to applying machine learning to end-to-end decompilation [3], [10], [33], [42], [45]. Recently, there have also been efforts to use large language models to improve the output of decompilers [39]. While there are no known machine learning methodologies tailored explicitly for control-flow structuring, end-to-end decompilation techniques inherently encompass some form of control-flow structuring.

3. Foundations and Definitions

While we assume familiarity with the key elements of CFGs, we include some explanations that may not be standard. Importantly, we provide a concise overview of the fundamental concepts that we consider essential for understanding the remainder of this paper. Moreover, we introduce some specific definitions that we utilized to simplify the subsequent sections and are thus essential for comprehension.

First, *entries* and *exits* (or *successors*) in a CFG define where control-flow enters and leaves subgraphs, while *sources* and *sinks* of edges represent the start and end points of control-flow between nodes. Here, *back-edges* point back to an earlier node in the CFG, often indicating the presence of loops within the flow, which is characteristic for *cyclic* graphs. In contrast, an *acyclic* graph lacks back-edges, representing linear code without loops.

As already mentioned, recovering high-level structures is not always straightforward and may necessitate the manipulation of the CFG, for example, by introducing gotos. We denote the C-structures into which subgraphs can be *translated* without such manipulations as *natural* C-structures. While each pattern from pattern-matching methods can be translated into a natural C-structure, this is not true for every subgraph handled by pattern-independent methods. However, the inherent properties of high-level structures ensure a clear sequence of code execution, thus necessitating the same for a subgraph to be resolvable into such a natural C-structure.

Accordingly, we define subgraphs that have a single-entry and either a single-successor or single-exit as *SISO* (single-in, single-out) regions. Additionally, because all C-structures have a single-entry, all cyclic subgraphs with multiple entries, also known as an *irreducible graph*, need to be converted into a *reducible graph*. Previous research demonstrated that code duplication can convert any irreducible graph into a reducible one [2], [20], creating a subgraph with one incoming edge from which all other nodes are accessible.

Finally, we note that any connected subgraph where at most one node has predecessors outside the region (denoted *entry* node) can be transformed into C-structures, if necessary, by manipulations as follows. For instance, any such CFG can be transformed using statements of the form `if (condition) goto Label;` to *virtualize* all edges, meaning that they are replaced by *guarded gotos*. This also implies that the same can be achieved for every such subgraph using only *structural variables*, which essentially encode the control-flow state of the program, enabling transitions between blocks based on the value of these structural variables [31], [53], [57].

Similarly, a well-structured CFG can be obtained by duplicating nodes until each node, excluding back-edges, has an in-degree of one or less, by replicating nodes with higher in-degree. Moreover, DREAM demonstrated that every single-entry subgraph can be translated to a C-structure solely with condition-duplication, through their multiple-exit restructuring for subgraphs with various successors and in combination with their general restructuring. However, because all of those strategies are very unlikely to yield a comprehensible output, no sensible decompiler would resort to such extremes.

4. Challenges in Control-Flow Structuring

Every control-flow structuring method, whether existing or prospective, must navigate a set of decisions or challenges, listed in Figure 1. Initially, some of these choices may appear straightforward and, depending on the objective, they may indeed be. For example, if readability is of no concern *whatsoever*, the control-flow graph can be transformed by using gotos to encode every edge of the graph (see Section 3). However, decompilers generally *do* have requirements, such as producing readable code to facilitate analysis, which significantly aggrevates making good choices when designing those approaches. Overall, we have identified four key decisions that need to be addressed during control-flow structuring:

1. Which subgraphs can be translated into a natural C-structure? *see Section 4.1*
2. How to handle subgraphs that cannot be translated into natural C-structures? *see Section 4.2*
3. Which subgraphs should be translated into C-structures in which order? *see Section 4.3*
4. Can the approach be considered practical for real-world scenarios? *see Section 4.4*

The first three decisions present significant challenges that directly affect the decompiler's output, where hasty choices might negatively influence the readability of the output. In contrast, the fourth decision deals primarily with the practical applicability of the approach. Naturally, these decisions can also greatly affect one another, such as the practicality limiting the available choices during the control-flow structuring phase. To the best of our knowledge, existing research for control-flow structuring usually discusses only a portion of these challenges, although all approaches must inevitably address all of them. We suggest that for a thorough understanding and comparison, future control-flow structuring approaches should carefully evaluate each of the four decisions.

Throughout the remainder of this section, we will examine each of the decisions along with the associated challenges we derived from related work, i.e. similarities/differences between existing approaches and unmentioned limitations. Furthermore, for each challenge, we will outline the corresponding limitations of current approaches. Finally, we will provide a summary of the impact of combining different choices on the overall output of decompilation approaches.

4.1. Definition of Resolvable Subgraphs

The first challenge of control-flow structuring arises from the division of current approaches into two categories: those that are pattern-independent and those that rely on pattern-matching. The existence of both suggests a difficulty in defining and identifying resolvable regions, i.e., in establishing what constitutes a *resolvable* subgraph or *region*, which is a subgraph that can be directly translated into natural C-structures. Depending on whether the control-flow structuring works in a pattern-independent way or with pattern matching, a definition of this or the generation of patterns for all resolvable regions becomes essential. Unfortunately, currently there is no universally practical solution for acyclic and cyclic regions to define a set of all resolvable regions or their characteristics.

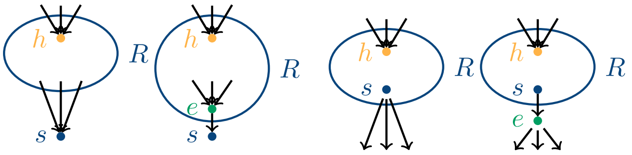


Figure 2: Transforming a single-successor to a single-exit region and vice versa; The node h denotes the single-entry and node s the single-successor or -exit, while e is a newly inserted, empty dummy node.

A fundamental requirement for regions to be resolvable, agreed upon by all current methods, is the presence of a single-entry point, as all natural C-structures specify the initial segment of code to be executed. Although natural C-structures inherently have a single-successor, indicating a clear post-structure execution path, a region must have either a single-successor or a single-exit to be resolvable. This is due to the fact that any single-successor region can be transformed into an equivalent region with a single-exit, and vice versa, by inserting an empty dummy node, as illustrated in Figure 2. However, not all subgraphs that have those two properties correspond to a natural C-structure. Next, we will highlight the challenges of defining acyclic and cyclic resolvable regions separately.

First, given an *acyclic* subgraph, determining whether it is resolvable is nearly unfeasible. Essentially, predefining all subgraphs that correspond to natural C-structures is practically impossible, a fact that is frequently demonstrated by modern decompilers utilizing pattern-matching. Despite there being a fixed number of natural C-structures, the potential variations resulting from compilers or obfuscators tampering with the control-flow graph, are endless. Therefore, even producing all subgraphs resulting from every natural C-structure does not account for variations that arise from various optimizations. For instance, a single natural C-structure can be compiled into subgraphs that differ from the trivial one, as demonstrated in Figure 3. Attempting to translate every possible *acyclic* subgraph that meets the SISO criteria into natural C-structures is impractical due to the necessity of aligning each subgraph with a natural C-structure. Even though this may be feasible for *some* subgraphs, determining whether a valid natural C-structure exists for a specific subgraph can be inherently difficult. In other words, the failure to easily translate the subgraph does not necessarily imply that there is no suitable natural C-structure; in fact, such a conclusion cannot be reached trivially. Additionally, this translation process might require the sequencing or nesting of multiple C-structures to properly align with the subgraph, or the computation of the conditions needed to access each node; both of these scenarios are either impractical or will likely yield suboptimal results.

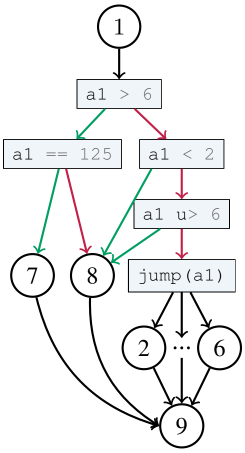
Second, for *cyclic regions*, determining which nodes belong to the loop structure is neither straightforward nor is it possible to make this decision optimally within the function context. Whenever there is a cycle in the control-flow graph, the control-flow structuring algorithm needs to translate a loop structure. Although every node that is part of a cycle must be included in a loop structure, it might be necessary to include nodes that are not part of any cycle into the loop structure to achieve a natural or an optimal C-structure. Besides, representing a loop by one circle is too restrictive, as loops are not necessarily disjoint, can be nested, or consist of multiple cycles due to continue-statements [37]. In contrast to acyclic regions, it is feasible to determine whether a cyclic region can be translated by making sure that the loop's back-edges conclude at the entry of a region that meets the described SISO criteria. However, a resolvable region does not always ensure that it will yield the best-possible output.

In conclusion, it seems impossible to find all acyclic resolvable subgraphs, and failure to translate a subgraph does not imply that it is not resolvable. Also, while we can determine whether a cyclic region is resolvable, this does not imply that choosing it produces the best output.

4.1.1. Limitations of Current Approaches Regarding the Definition of Resolvable Regions. To the best of our knowledge, all existing control-flow structuring methods have limitations regarding the definition of both acyclic and cyclic regions. Generally, there are two distinct ways in which approaches manage *acyclic regions*. First, pattern-matching techniques implicitly define a resolvable region as a subgraph that fits one of many predefined patterns [7], [13], [38], [51], with each pattern being translated into a natural C-structure. However, as already noted, obtaining a complete set of patterns is impractical, as demonstrated by even state-of-the-art decompilers repeatedly failing to handle certain subgraphs. As a consequence of missing patterns, such approaches will have to cope, e.g. by emitting gotos, with suboptimal decompiler output leading to potential readability issues. For instance, at the time of writing, the control-flow graph in Figure 3a is not translated into a natural C-structure by Ghidra and SAILR (see Figures 3c and 3d), despite the existence of one. It is clear that the resulting output does not utilize the available C-structures as effectively as it could, opposed to pattern-matching approaches such as dewolf in Figure 3b.

Second, in contrast to the above, pattern-independent methods employ explicit definitions for resolvable regions, meaning they consider all subgraphs that possess a single-entry and a single-successor [35], [68], [73]. However, as mentioned previously, not every such subgraph is actually resolvable; some regions require modifications before being translated into C-structures. Therefore, since these methods do not differentiate between resolvable and non-resolvable acyclic regions, they may unconsciously suffer from the possible negative effects of the transformations needed to make a non-resolvable region resolvable. For example, DREAM treats both resolvable regions and regions where conditions are duplicated to make them resolvable identical, while simultaneously complicating the output of the latter. Similarly, Binary Ninja translates every single-entry, single-successor subgraph into C-structures using code-duplication, condition-duplication, and gotos if necessary. Finally, rev.ng duplicates code and introduces structural variables in their preprocessing to achieve a well-structured graph that can then be translated into natural C-structures. All these methods introduce additional complexity and could likely be optimized by explicitly considering the differences between resolvable and non-resolvable regions.

For *cyclic* regions, all existing pattern-independent approaches only consider a single-entry, single-successor



(a) CFG

```
test1(a1){
  printf("1");
  switch(a1){
    case 2:
      printf("2");
      break;

    case 6:
      printf("6");
      break;
    case 125:
      printf("7");
      break;
    default:
      printf("8");
  }
  v = printf("9");
  return v;}
```

(b) dewolf

```
test1(a1){
  printf("1");
  if (a1 > 6){
    if (a1 == 125){
      printf("7");
      return printf("9"); }
  L1:
    printf("8");
    return printf("9"); }
  if (a1 < 2) {
    goto L1; }
  switch (a1){
    case 2:
      printf("2");
      break;

    case 6:
      printf("6");
      break;
    default:
      goto L1; }
  return printf("9"); }
```

(c) Hex-Rays

```
test1(a1) {
  printf("1");
  if (a1 <= 6) {
    switch (a1) {
      case 2:
        printf("2");
        return printf("9");

      case 6:
        printf("6");
        return printf("9");
      default:
        printf("8");
    }
    return printf("9"); }
  else if (a1 != 125) {
    printf("8"); }
  else {
    printf("7");
    return printf("9"); } }
```

(d) SAILR

Figure 3: An exemplary CFG that does not obviously match a natural C-structure. While the pattern-independent approach dewolf recovers a natural C-Structure, namely a switch, as in the source code, Hex-Rays duplicates the return and uses a goto for the default, whereas SAILR duplicates the default and the return to avoid gotos. In contrast, Ghidra uses multiple gotos by not duplicating any code. All outputs have been standardized to highlight the structural differences.

loop-region, thereby overlooking loop-regions with a single-exit. Furthermore, they greatly restrict their options by treating loops within the same definition of resolvable regions, which should be considered separately, as we will discuss in more detail in Section 4.3. Pattern-matching methods also constrain themselves by only including loop-nodes in resolvable cyclic regions since the introduction of dcc [13], [15]. Although there have been extensions over the years that allow additional nodes within the region (cf. [7]), none have yet considered all possible resolvable regions. For example, the Phoenix decompiler, which also underpins SAILR, presumes the loop-successor to be the successor of either the back-edge source or sink (loop-head). Observations also indicate that Ghidra and Hex-Rays do not adequately extend each loop, leading to the introduction of gotos despite the existence of a resolvable region as also illustrated by Figure 6b. In summary, both method-types could benefit from revised, explicit or implied, definitions of resolvable and non-resolvable (a)cyclic regions and their proper consideration.

4.2. Handling Non-Resolvable Subgraphs

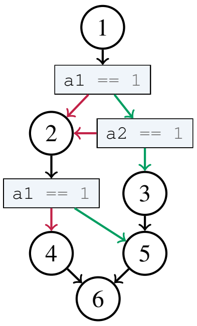
The second challenge in control-flow structuring arises from the existence of the two categories of approaches: those that either emit or avoid gotos. This extends to the broader challenge of managing non-resolvable subgraphs, primarily due to the difficulty in determining which strategy eventually yields the best results.

Unfortunately, it is not possible to translate every subgraph into natural C-structures without prior modification. This is often due to various compiler optimizations or gotos in the original source code, which can produce highly unintuitive binary code. We refer to regions that can be made resolvable with certain modifications as *conditional regions*. To the best of our knowledge, there are four viable options from which any approach *must* choose, and which can be used to translate every subgraph (see Section 3): introducing gotos, duplicating code or conditions, and adding structural variables. It is practically *impossible* to determine which option results in the best output for the entire function. Even when able to impeccably rate local decompiler output quality, this would not necessarily result in an optimal function output. In addition, each option adds an additional layer of complexity compared to natural C-structures. Figure 4 shows a exemplary control-flow graph that is translated with different options, resulting in clearly different outputs. In the following, we will discuss each of the available options and their impact on the output. It is important to note that simply allowing all options does not significantly ease the overall problem, because choosing the best option per situation becomes impractical (see Sections 4.3 and 4.4).

The simplest and most commonly used option to handle non-resolvable subgraphs are *gotos*. While gotos can make *every* subgraph resolvable, they should obviously not be applied to *every* subgraph to make appropriate use of the available C-structures and produce a high-level output. Still, there are valid reasons to utilize them, such as exiting an inner loop, but they can significantly aggravate manual analysis, especially when causing jumps to entirely different sections of the code.

In contrast, a more intuitive option for achieving a resolvable region is *code duplication*, assuming that non-resolvable regions arose from compiler optimizations or obfuscation techniques. Essentially, duplicating small pieces of code, such as return statements, will probably not have a negative impact on the output quality. However, when limiting the options to duplication alone, we might also need to replicate entire C-structures, such as loops, which could substantially increase the analysts' workload.

Although *duplicating conditions* might initially seem unintuitive, it can already be considered in use when defining resolvable regions solely with the SISO criteria;



(a) CFG

```
test2(a1, a2) {
  printf("1");
  if (a1==1 && a2==1) {
    printf("3"); }
  else {
    printf("2");
    if ( a1 != 1 ) {
      printf("4");
      goto L1; } }
  printf("5");
L1:
  printf("6");
  return 0LL; }
```

(b) Hex-Rays, Ghidra

```
test2(a1, a2) {
  printf("1");
  if (a1==1 && a2==1) {
    printf("3"); }
  else {
    printf("2"); }
  if (a1 == 1) {
    printf("5"); }
  else {
    printf("4"); }
  printf("6");
  return 0L; }
```

(c) DREAM [73], dewolf

```
test2(a1, a2) {
  printf("1");
  if (a1!=1 && a2!=1) {
    printf("2");
    if (a1 == 1) {
      printf("5"); }
    else {
      printf("4"); } }
  else{
    printf("3");
    printf("5"); }
  printf("6");
  return 0; }
```

(d) Binary Ninja

Figure 4: An exemplary CFG with a natural C-structure. Hex-Rays and Ghidra use a goto, while SAILR and Binary Ninja duplicate code (basic block 5), and DREAM organizes the control-flow without either. If the value of `a1` in basic block 2 was changed, DREAM's output would be rendered invalid - in contrast to the recovered C-structures of the other decompilers. All outputs have been standardized to highlight the structural differences.

see [73, Figure 3 and 7]. Similarly to code duplication, repeating simple conditions can enhance readability. However, relying solely on this option can result in extensive conditional regions, creating long and complex conditions that are difficult for an analyst to comprehend and to compute during decompilation (see Section 4.4). Also, they may inadvertently cause undesirable side-effects [73].

Finally, when introducing *structural variables*, edges are redirected by introducing new conditions [69], [73]. This not only adds complexity by additional conditions, but usually also increases the nesting depth. Furthermore, the conditions themselves often lack meaning, raising doubts about their usefulness in helping the analyst comprehend the output. Nonetheless, using conditions to direct the control-flow can still result in more structured code compared to jumping to an arbitrary location with gotos.

In conclusion, determining the *best* choice of option(s) for transforming a conditional region into a resolvable to achieve an optimal overall output appears infeasible because every option has its merits and drawbacks. Presumably, approaches should only exclude available choices when justified by valid reasons.

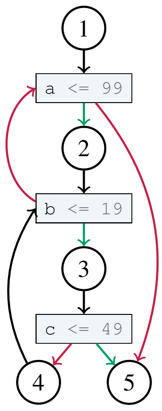
4.2.1. Limitation of Current Approaches Regarding Handling Non-Resolvable Subgraphs. The main issue with current control-flow structuring approaches is that they typically only take into account a very limited selection of the above-mentioned options to make a conditional region resolvable. Although some adopt more than one option, they generally employ these additional options very sparingly and only for particular situations. As a result, the abilities of each method are limited as the majority of potential options are ignored from the beginning.

Most pattern-matching techniques for structuring control-flow predominantly, if not all, depend on gotos to manage non-resolvable regions [7], [38], [51]. Another prevalent option is code duplication, such as Hex-Rays copying return statements or SAILR duplicating code that meets specific criteria [4]. However, since they only address the problem locally, they occasionally disrupt natural C-structures through code duplication, as already illustrated in Figure 3d. Furthermore, it can be observed in Figures 4b and 4d that those options do not necessarily lead to the best output. Overall, we find that being overly stringent with the use of these options can lead to needless gotos or suboptimal readability in the decompiled output.

In contrast to using gotos and duplicating code, the pattern-independent DREAM approach mainly duplicates conditions to translate each single-entry and single-successor region into C-structures, according to their definition of resolvable regions. Unfortunately, prohibiting any code duplication or gotos sometimes results in very large resolvable regions. The consequences are very long and complex conditions or unnecessarily complicated structured, such as observable in Figure 5b. These conditions can be extremely difficult for human analysts to understand, greatly aggravating manual analysis and reducing analysis efficiency. To translate multiple-entry and multiple-exit loop regions, DREAM additionally uses structural variables to redirect the control-flow. However, this is not desirable for all of those cases, such as when the control-flow exits an inner loop, where using gotos would be far more suitable than structural variables. An example of such a control-flow graph is provided in Figure 5.

In an attempt to address some of DREAM's limitations, rev.ng chooses to use *structural variables* and *code duplication* to manage non-resolvable regions. Nonetheless, a major issue with rev.ng is its overly aggressive duplication, which can result in the needless duplication of entire structures, such as loops, thereby significantly increasing the cognitive burden on analysts. In summary, both DREAM and rev.ng impose unnecessary constraints on themselves by not utilizing all available options.

To the best of our knowledge, the only approach that implements three of the four options is Binary Ninja. Even though they also translate each single-entry, single-successor region, they further permit code and condition duplication as well as the use of gotos. Despite employing these options to generate a subgraph where each node's in-degree is at most one, which facilitates the resolving into natural C-structures, they still limit themselves with overly strict criteria for considered subgraph. Specifically, they overlook conditional regions by focusing solely on single-entry and single-successor regions, ignoring regions that may arise through duplication. Also, similar to others, they might duplicate more code than necessary because they do not optimize the output on a function level but rather focus on locally optimizing the current subgraph.



(a) CFG

```
test3() {
  printf("1");
  while (a <= 99) {
    printf("2");
    while (true) {
      if (b > 19) {
        exit = 0;
        break; }
      printf("3");
      if (c <= 49) {
        exit = 1;
        break; }
      printf("4"); }
    if (exit != 0) {
      break;  }  }
  var = printf("5");
  return var; }
```

(b) DREAM [73], dewolf

```
test3() {
  printf("1");
  while (a < 100) {
    printf("2");
    while (b < 20){
      printf("3");
      if (c < 50) {
        goto L1; }
      printf("4"); } }
L1:
  printf("5");
  return; }
```

(c) Ghidra, Phoenix

```
test3() {
  printf("1");
  while (a <= 99) {
    printf("2");
    for (b <= 19) {
      printf("3");
      if (c <= 49) {
        return printf("5"); }
      printf("4"); } }
  return printf("5"); }
```

(d) Binary Ninja, Hex-Rays, SAILR

Figure 5: An exemplary CFG without a natural C-structure. Ghidra opted to use goto statements, whereas DREAM utilizes its multiple-exit restructuring, and Binary Ninja, Hex-Rays, and SAILR duplicate the return as well as the print-statement. All outputs have been standardized to highlight the structural differences.

4.3. Subgraph Resolving Order

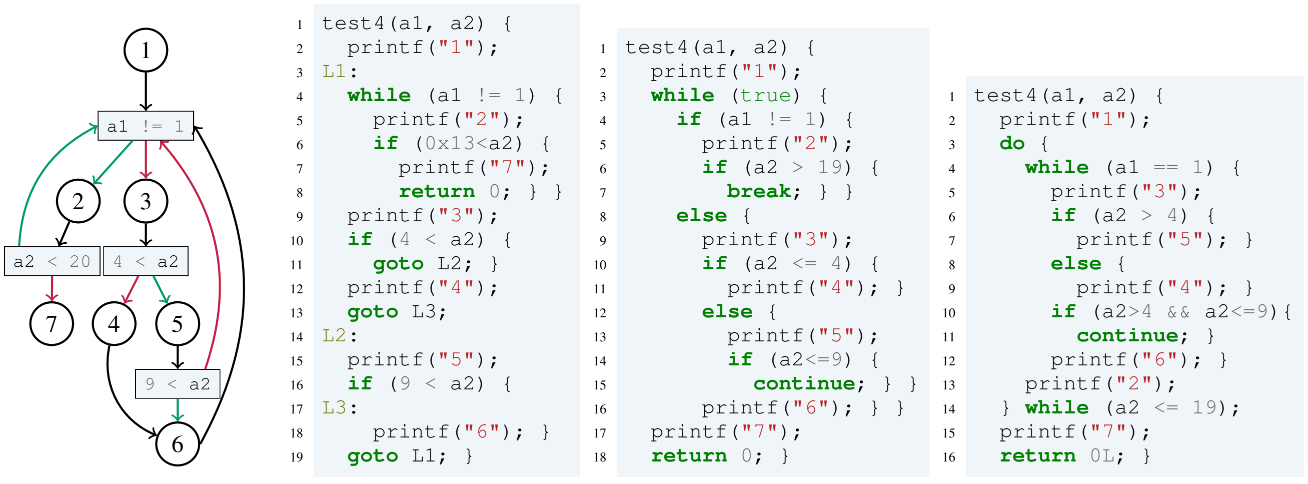
The third challenge in control-flow structuring originates from definitions of a resolvable region that often produces multiple options or patterns for a given subgraph. As a result, the third challenge is deciding among these potential regions, or more broadly, determining the order in which the regions should be translated, or converted into suitable C-Structures. Since the conversion of one region affects other regions, this requires choosing between different regions and/or ranking them by priority. Currently, all existing methods [4], [7], [15], [35], [72], [73] agree that inner structures should be processed before outer structures. By this, all successors are processed and potentially structured before examining each node itself. For example, recognizing an if-(else-) becomes easier once the branches of a condition are structured.

Traversing inner before outer structures can be implemented by visiting nodes in post-order to select the next resolvable region entry, as consistently implemented by existing methods. Here, there are typically multiple choices for each region-entry within both acyclic and cyclic regions, posing the challenge to choose between them. In cyclic regions, it is crucial to determine which cycles with back-edges to the region-entry should be prioritized, as not all cycles can be restructured without affecting another cycle. For example, in Figure 6, the cycle containing 2 can only be translated into a SISO-region when the entire cycle, including nodes 3 to 6, is also considered part of the loop. However, this SISO-region does not correspond to a single C-construct, whereas the cycle containing the nodes 3 to 6 corresponds to a while-loop. Thus, it is not obvious which cycles should be prioritized to achieve the best output, without computing all possibilities. As demonstrated in Figure 6d, the example can in fact be translated in two nested loops: a while-loop (containing nodes 3 to 6) nested within a do-while-loop with node 2. Still, Phoenix and SAILR only restructured one infinite loop, while Ghidra and Hex-Rays resort to emitting gotos, as illustrated in Figures 6b and 6c.

Although Section 5 demonstrates that calculating one particular resolvable region for a given entry can be sufficient to achieve the optimal order for (a)cyclic regions, it remains difficult to do so practically for acyclic regions: As discussed in Section 4.1, it is impossible to say with certainty for each subgraph whether it corresponds to a resolvable region. Furthermore, it is not straightforward to determine whether the conditional regions should be translated or skipped in favor of the next entry-node where a resolvable region is available. Even for nodes with existing resolvable regions, other conditional regions might yield a better outcome, adding to the complexity of the problem. For example, by duplicating return-statements, the complexity of a program can be reduced drastically. The challenge lies in identifying which regions, including both resolvable and conditional regions, provide the best result. For instance, as already shown in Figure 3, premature duplication can potentially undermine structures. More precisely, while it might locally appear beneficial to duplicate node 8 to obtain a switch with a default case, this will ultimately disrupt the existing switch structure because node 7 is also a switch case. In fact, for this example, not resolving the conditional region led by `if(x >u 5)` results in a better decompiler output for this function.

Determining an optimal resolving sequence is inherently challenging, as it requires the decision to either translate a given entry-node's resolvable or conditional region, or to skip it. First, it is unclear which regions need to be computed in order to make an informed decision, or whether regions can be considered irrelevant from the outset. Second, given that any connected subgraph where one node reaches all other nodes is considered a conditional region, there may be an exponential number of such regions, rendering an exhaustive consideration of them impractical. Finally, the number of possibilities is additionally increased because each conditional region can be translated in multiple ways.

In conclusion, locally deciding whether to translate a resolvable region or a conditional region, as well as choosing one, seems nearly impossible because every decision can influence future choices; only because a decision is locally optimal does not imply that it also leads to overall optimal function output.



```
test4(a1, a2) {
  printf("1");
L1:
  while (a1 != 1) {
    printf("2");
    if (0x13<a2) {
      printf("7");
      return 0; } }
  printf("3");
  if (4 < a2) {
    goto L2; }
  printf("4");
  goto L3;
L2:
  printf("5");
  if (9 < a2) {
L3:
    printf("6"); }
  goto L1; }
```

```
test4(a1, a2) {
  printf("1");
  while (true) {
    if (a1 != 1) {
      printf("2");
      if (a2 > 19) {
        break; } }
    else {
      printf("3");
      if (a2 <= 4) {
        printf("4"); }
      else {
        printf("5");
        if (a2<=9) {
          continue; } }
      printf("6"); } }
  printf("7");
  return 0; }
```

```
test4(a1, a2) {
  printf("1");
  do {
    while (a1 == 1) {
      printf("3");
      if (a2 > 4) {
        printf("5"); }
      else {
        printf("4"); }
      if (a2>4 && a2<=9){
        continue; }
      printf("6"); }
    printf("2");
  } while (a2 <= 19);
  printf("7");
  return 0L; }
```

(a) CFG (b) Ghidra (c) SAILR, Binary Ninja, revng (d) dewolf, DREAM [73]

Figure 6: An exemplary CFG with two loop regions being structured by different decompilers. In constrast to the others, Ghidra employed gotos, while dewolf and Hex-Rays structure the loop-structures of the source code successfully by recovering two nested loops with meaningful conditions. On the other hand, SAILR and Binary Ninja restructure the two loops as one endless-loop. All outputs have been standardized to highlight the structural differences.

4.3.1. Limitations of Current Approaches Regarding the Order in which Subgraphs are translated. A major drawback of current control-flow structuring methods, despite acknowledging the impossibility of computing *all* resolvable and conditional regions, is that they only compute *a single* region for each region-entry, without providing an explanation. Moreover, their reliance solely on patterns or domination for computation further limits their ability to find the regions that yield the best output. Admittedly, the fact that only one region is computed in the first place makes the selection of the next region to translate relatively straightforward. Nevertheless, this often results in less optimal outcomes, making it an effective limitation since it ignores other potentially better-fitting regions in preference to the one calculated.

In particular, traditional pattern-matching approaches [7], [13], [38], [51] try to find at least one resolvable acyclic region from their pattern-set, and use heuristics when none can be found. With these heuristics, they determine which edges to virtualize, as done by Hex-Rays with gotos, or by occasionally duplicating instructions when no resolvable region is present. In contrast, SAILR indirectly identifies extra conditional regions during their de-optimization phase, but still focuses on computing only one extra conditional region for each node. However, as their primary goal is to revert compiler optimizations rather than enhancing readability, this might be perfectly deliberate. Typically, these methods employ gotos to manage non-resolvable regions [7], [38], [51], though we have noticed that Hex-Rays sometimes duplicates return statements to circumvent the use of gotos. In general, the stringent limitation on one resolvable and conditional region significantly limits the output quality of the decompiler in certain cases.

Regarding pattern-independent methods, the DREAM approach aims to identify a single-entry, single-successor region for each node, treating it as the single-entry point. Thus, they consistently compute only one acyclic region per given entry, which includes all nodes dominated by the entry node of the region. However, all nodes dominated by the region's entry might not always be a resolvable region, although a subset could be. As a result, they can overlook resolvable regions even when they exist. Furthermore, they may also fail to detect smaller conditional regions that could be resolvable through methods such as condition duplication. Here, dewolf makes a slight improvement by identifying nodes within the domination set that divide it into two connected components, aiming to locate a smaller SISO-region if it exists.

Similarly, Binary Ninja considers single-entry, single-successor regions for every node. Despite the uncertain details of their method, it seems that they always *attempt* selecting the smallest possible such region. Using also domination techniques, rev.ng additionally relies on extensive preprocessing, involving significant duplications. This ensures that the domination set of the entry is resolvable and minimal when analyzing the nodes in post-order. To choose nodes for duplication, they apply heuristics and *sometimes* calculating one alternative region to reduce duplication (see their *Untangling* [35]). Overall, these methods overlook potential alternative strategies to discover different resolvable regions, unnecessarily restricting themselves regarding output quality.

In contrast to acylic regions, every cyclic region should be translated to ensure adequate handling of loops, necessiting approaches to compute a conditional cyclic region if no resolvable region exists. Here, for both resolvable and conditional cyclic regions, each approach must determine which loops should be part of the same region and be translated first. Unfortunately, many methods make simplistic assumptions about which loops to include without thorough consideration, usually being overly restrictive and not yielding the best possible result. For instance, the Phoenix decompiler considers only a single cycle and selects the smallest cycle as the *initial-loop region*. Subsequently, they perform a *loop-refinement* that extends the initial loop-region and tries to prevent multiple exits by incorporating nodes that are dominated by the region entry and reach the loop successor. This refinement can potentially include additional loops to the cyclic region. A major restriction here is that the loop successor is the immediate successor of either the source or sink of the back-edge defining the initial-loop region.

On the other hand, DREAM initially defines a loop region as encompassing all loops whose back-edges converge to the same sink. Their loop-structuring rules enable the detection of nested loops, although they are considered as one cyclic region. However, considering the inner-loop first would guarantee that it is structured as an inner loop. During the loop refinement, if the initial loop-region has more than one successor, those successors are incorporated into the loop-region if the region entry dominates the successor. This procedure continues until only one successor remains or no further additions are possible. Unfortunately, this extension is suboptimal and might not identify existing single-successor regions and could lead to a cyclic region with more successors than initially.

Finally, the initial loop calculation of rev.ng may introduce redundant structural variables. They aim to sort the loop-regions and combine two loops if they intersect and neither is a subset of the other. However, their illustrations in Figures 2 and 3 are translated into a single cyclic region, introducing structural variables to create a resolvable region [35]. However, in both scenarios, the output is unnecessary complex because a subset of the nodes is a resolvable cyclic region. Moreover, their loop-refinement is overly rigid, as it only adds nodes dominated by the region-entry and its predecessor within the region.

4.4. Practicability

The runtime limitations encountered in current control-flow structuring methods, such as those of dewolf and rev.ng, present the fourth challenge: *practicability*. When considering methods for control-flow structuring with an emphasis on readability, the general objective is to identify optimal structures that facilitate analysis. Although we acknowledge that a program does not need to decompile instantly, the decompilation process, including control-flow structuring, *must* remain practically feasible. This typically means that the decompilation time of a binary should be proportional to the time that an analyst requires for examination. Within a constrained timeframe, the task is to achieve the most readable output possible while maintaining feasibility. As pointed out in the previous challenges, obtaining the best result is mostly *unfeasible*, as the global consequences of local choices are generally unknown at the time of making those decisions.

In theory, control-flow structuring can be optimally solved when only considering the first three challenges, i.e., we would always be able to generate the best possible output for a given metric. Concretely, a control-flow structuring technique can be visualized as a decision tree: The nodes of the control-flow graph would still be examined in post-order, but for each node, we would compute all resolvable and conditional regions associated with that entry. The decision tree proceeds either with one of the regions or decides not to translate a region with this entry for acyclic regions. Subsequently, we move to the next node in post-order for each current leaf node in the decision tree. Given that the number of conditional regions is already exponential, the size of this decision tree is vast and thus is the runtime.

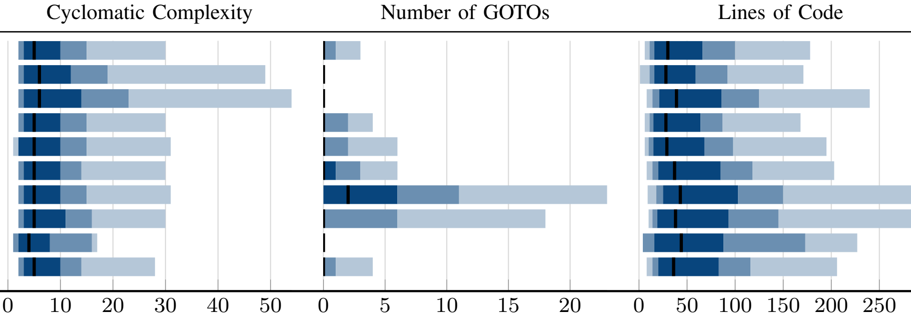
At the time of writing, 70.46% of all samples from Malpedia [55] have at least one function with a minimum of 100 nodes (and at least 500 nodes for 10.20% of the samples). Considering a function graph with 100 nodes, if the decision tree contains 2^{100} leaves, it would take a minimum of 10^{15} days to complete the computations for all of them, even if each computation takes only 0.1 nanoseconds, for example when using a 10 GHz CPU capable of performing one such computation per cycle. Clearly, such an extensive runtime is impractical for decompiling a single function in a practical setting.

In conclusion, although a theoretical approach could always yield the perfect overall output, such strategies seem impractical. Consequently, maintaining practicability seemingly necessitates some heuristics and compromises.

4.4.1. Limitations of Current Approaches Regarding the Practicability on Real-World Samples. Generally, pattern-matching methods do not face significant issues with practicability. We assume that this is a conscious decision because they are intended to function effectively in practice. Thus, they are embedded in numerous decompilers that are frequently utilized by malware analysts. While they might not deliver the best output due to their restrictions regarding the previous challenges, they generally manage to generate output for most functions within reasonable speed.

In stark contrast, some pattern-independent approaches, such as DREAM, may easily encounter runtime problems when faced with a complex control-flow graph structure. Only allowing the duplication of conditions when there is no acyclic resolvable region can lead to extremely large areas, sometimes exceeding 100 nodes, which is not unusual in malware decompilation. In such regions, calculating the conditions required to reach each node or arranging them into C-structures becomes impractical. Consequently, the extensive logic operations and simplifications result in these functions failing to decompile even after several hours. This further proves that neglecting too many of the strategies covered in Section 4.2 limits methodologies in various ways.

However, pattern-independent methods do not necessarily encounter practicality issues. For instance, both rev.ng and Binary Ninja circumvented runtime issues by design. In the rev.ng approach, the preprocessing might result in overly lengthy decompilation output, yet it simplifies the matching of C-structures and facilitates control-flow structuring significantly. Moreover, their preprocessing is not overly time-consuming as they only duplicate nodes and consider up to two alternatives for a single region, making decisions on a local basis. By this, they try to avoid not generating results within a reasonable time frame. Similarly, Binary Ninja also makes local decisions when restructuring a region. It is unclear if they consider multiple regions, but they appear to use heuristics to determine which options to use when resolving C-structures. Additionally, they apparently prioritize their practicality to ensure generating results for the majority of functions. Their earlier implementation also demonstrated that DREAM can be made practical by employing gotos when conditions become overly complex.



	Timeout	Failed
Binary Ninja	16.1%	2.6%
dewolf	21.1%	5.3%
DREAM	19.5%	20.8%
Ghidra	1.2%	24.3%
Hex-Rays	2.1%	7.7%
Phoenix	20.5%	10.4%
Radare	0.5%	22.8%
RetDec	1.2%	36.2%
rev.ng	65.8%	31.6%
SAILR	20.6%	10.3%

TABLE 2: Assessment results. The interval plots visualize the distribution of function metrics, where ■ represents the central 50% of values, ▨ covers 70%, and □ includes 90%. The black vertical line within each plot indicates the median.

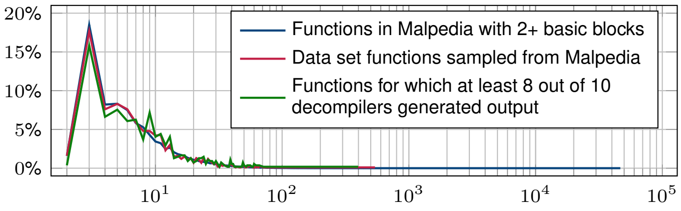


Figure 7: Function sizes (num. basic blocks) distribution.

4.5. Decompiler Output Assessment

To strengthen the arguments we made in previous sections and showcase some limitations with real-world examples, we employed several decompilers, with vastly different design decisions, that were operable within a reasonable timeframe on a fixed dataset of binary functions and applied various metrics used in existing research.

For our data set, we randomly sampled 1000 functions from all functions recognized by smda [54] in all unpacked samples from Malpedia [55]. Notably, we excluded functions with a single basic block since they do not require any structuring. We also aimed to achieve a level of representativeness concerning function size, assessed by counting basic blocks, as instruction counts within blocks are insignificant for structuring. Figure 7 compares the function sizes of our data set with those 533 decompiled by at least 8 of the 10 decompilers and entire Malpedia.

Table 2 presents our assessment results of the decompilers mentioned in Table 1. This includes the percentage of functions that resulted in errors or yielded no output, as well as the percentage of functions that generated no output within a five-minute period per function, a time-frame we consider reasonable for decompiling a single function. As highlighted previously in Section 4, implementing a timeout was crucial due to the impracticality of expecting every decompiler to efficiently handle large functions. Additionally, it presents the distributions of three commonly used metrics for the outputs from functions that at least eight decompilers processed successfully: cyclomatic complexity, number of gotos, and lines of code. Although these are not fully sufficient to assess decompiler output quality, they are commonly used and provide a simple assessment. We refrained from measuring source code distance because such data was unavailable for Malpedia.

While our findings indicate that the chosen metrics yield results that are relatively consistent across most decompilers they also align with several limitations discussed in prior sections. For example, pattern-independent approaches are generally more prone to timeouts, except for phoenix, sailr, dream – implemented in Python – and rev.ng. Furthermore, while DREAM and dewolf successfully eliminate gotos, they tend to increase cognitive complexity, likely due to adding structural variables. Conversely, Binary Ninja effectively reduces gotos with minimal impact on cognitive complexity, presumably by adding condition duplication, when compared to Hex-Rays and Ghidra. Interestingly, radare and RetDec produce significantly more gotos and lines of code, suggesting fewer recovered structures and a more assembly-like output, though structuring may not be their primary focus.

We point out that setting a timeout for each function has certain drawbacks due to the fact that decompilers conduct varying levels of analysis on the entire binary before decompiling specific functions, which we could not consistently isolate. Importantly, rev.ng only allows for the decompilation of the full binary, which might result in extra timeouts or failures. Lastly, the assessment provided is not designed to accurately compare decompilers overall; we did not evaluate either correctness or the impact of various data-flow analyses, for example. Moreover, a proper comparison of different control-flow structuring algorithms is not feasible as it would necessitate their implementation within the same decompiler framework.

4.6. Limitations Summary

We conclude that current methods probably restrict their output quality by not considering the whole picture. On the one hand, pattern-matching approaches prioritize practicality, which is justified in principle, since analysts need *some* output. But, they could adopt more flexible region definitions, utilize more options to convert conditional into resolvable regions, or consider multiple regions.

To address this, SAILR considers alternative conditional regions to improve the output quality in some instances. However, although their approach might result in outputs that more closely match the original source code for some highly optimized projects actually using gotos, it does not necessarily align with our primary goal of producing the most comprehensible output. Particularly in the context of malware, the binary cannot be assumed to come from well-written source code and might even have been obfuscated prior to compilation. Moreover, not all compiler optimizations that might have been applied to a given binary can be identified post compilation [28].

On the other hand, and in contrast to pattern-matching, DREAM and similar methods partially neglect practicality. Although they challenged that gotos are the sole solution for control-flow structuring, they still limit themselves by exclusively allowing condition duplication. Furthermore, they risk creating very lengthy regions, potentially causing the decompiler to not produce any output. Likewise, rev.ng may lead to very lengthy output due to their duplication approach, potentially of entire structures such as loops. Although this might address practicality issues without using gotos, it can notably enlarge the output size.

Ultimately, we regard Binary Ninja's method as the sole approach that leverages nearly every option for managing non-resolvable subgraphs. Consequently, they do not restrict their ability to obtain resolvable regions as much as others. However, they only calculate one instead of other potential resolvable and conditional regions, possibly overlooking chances to minimize code duplication.

5. Discussion

Although we have already recognized the interdependent nature of the challenges of Section 4 and the inherent difficulties of control-flow structuring, we identify potential for significant enhancements in various elements of existing approaches. Therefore, we propose concepts and potential strategies that future control-flow structuring techniques may adopt for improvement. Primarily, we show a drastic reduction in the number of regions requiring calculation, even when taking into account all discussed options, thereby notably improving the solution-space for future research. Finally, we briefly discuss the trade-offs each approach must consider when leveraging on our suggested improvements. Altogether, the concepts in this section can serve as a preliminary framework for a control-flow structuring method, customized for specific goals, providing there is a willingness to consider certain heuristics or compromises.

Generally, we believe that future approaches should particularly focus on the order in which the regions are resolved while still maintaining practicability. This stems from our proposal to address the first two challenges by being as general as possible when defining resolvable regions and handling non-resolvable subgraphs, thus shifting the complexity to challenges three and four: practically optimizing the resolving order. Theoretically, this problem could be solved optimally given a metric assessing every possible outcome; but, this is impractical due to the exponential increase in computation time. Hence, a practical solution requires trade-offs to ensure feasible decompilation times while recognizing the limitations of each design decision. Without such crucial trade-offs, an approach is likely to fail in producing output for large functions, as illustrated by some decompilers producing goto-free output but lacking practicality due to constraints on managing non-resolvable regions. In the remainder of this section, we explain how future research could reduce both the challenge of defining resolvable regions and handling non-resolvable regions to the challenge of optimizing the resolving order; we therefore focus on maintaining practicality while accomplishing this. Ultimately, however, some issues still persist that research has not yet been able to solve, necessitating certain trade-offs.

5.1. Definition of Resolvable Regions

The examples in the previous sections, as seen in Section 4 and Figure 3, indicate that excluding regions that can be translated into natural C-structures may lead to suboptimal results. Consequently, we propose defining resolvable regions as broadly as possible, which means that every SISO region that can be translated into a natural C-structure should be considered resolvable. Although this admittedly complicates the decision between resolution orders slightly and might influence practical feasibility, we contend that it remains beneficial to ensure that natural C-structures are not overlooked. Moreover, the additional computational load should still be manageable, as we demonstrate with our following approach to handle the necessary computations of SISO-regions.

Specifically, we demonstrate that to achieve an optimal output, future methods only need to compute one particular resolvable region for a given single-entry node, regardless of the potential existence of many such regions. This approach significantly reduces the number of resolvable regions and the total number of possibilities that future methods must consider.

Observation 1. For a given entry-node, considering only one resolvable (a)cyclic region still guarantees an optimal decompiler output.

Essentially, we can establish a *total preorder* on the resolvable regions for a given entry node, based on the principle that inner-regions should always be translated before outer-regions. Now, for each entry, selecting the region based on this order guarantees the best output among all resolvable regions with the same single-entry, as shown in Section B.1.

However, computing even a single *acyclic* resolvable region remains a challenge, particularly to decide whether a subgraph is resolvable. Theoretically, it can be established that a resolvable region, which is also a SISO-region, cannot be translated if it cannot be converted into a natural C-structure. Nevertheless, for a practical approach, one probably must employ *some* kind of heuristic, since the inability to readily convert a subgraph into a natural C-structure does not necessarily indicate the absence of an appropriate one, as previously addressed in Section 4.1. For example, one could try to transform the subgraph into a natural C-structure similar to DREAM or set a time limit to search for a natural C-structure. In both scenarios, one would have to assume that the region is not resolvable if no natural C-structure is found.

Unfortunately, even minimal SISO-regions can become quite large, particularly in case of malware, making it increasingly unlikely and costly to test whether the region can be translated into a natural C-structure. In such cases, we would recommend not trying to find a natural C-structure to remain practicable until future research yields better heuristics. Conversely, for *cyclic* regions, it is possible to compute such a region in polynomial time if it exists: For each loop whose head is the current region entry, i.e., each loop whose back-edge sink is the head, compute all smallest *cyclic* SISO-regions that contain it. Afterwards, select the best region regarding the total preorder mentioned above among these cyclic regions.

5.2. Handling Conditional Regions

Given that a resolvable region cannot always be identified, it is usually necessary to consider how to convert conditional regions into resolvable regions. As shown previously in Figures 4 and 5, excluding viable options can potentially restrict the quality of the output. Therefore, we recommend using all four available choices: duplicating both conditions and code, using structural variables, and introducing gotos as inclusive as possible when considering conditional regions. However, an exception exists for methods with goals that inherently rule out certain options, such as producing goto-free decompiler output. Here, it may actually be necessary to exclude a subset of the options - regardless of the impact on the final output quality to achieve the desired goal.

In addition, we suggest that future studies aim to discover new options for transforming conditional regions into resolvable regions. In particular, the authors of SAILR mention deduplication to improve readability in cases where compilers have duplicated code [4], which could be considered an additional option in this context. However, we believe that compiler de-optimizations should be addressed separately to not further complicate control-flow structuring; if necessary, deduplication and other de-optimizations should probably be applied before.

Unfortunately, considering more options naturally increases the number of regions; essentially, each connected subgraph, characterized by a single node having a pathway to every other node within the subgraph, is a conditional region. To ensure an optimal resolution sequence, all such subgraphs must be considered, which quickly makes the method impractical. Therefore, we propose additional heuristics to effectively limit the scope to a subset of all subgraphs. We believe that this allows for a somewhat practical approach without overly restricting capabilities by making the following observation:

Observation 2. Considering only conditional subgraphs of minimal SISO-regions still guarantees an optimal decompiler output, excluding outputs duplicating the single-entry or the single-successor/exit.

This observation stems from the principle that control-flow is restricted to entering SISO-regions via the single-entry and exiting through the single-successor/exit. For a more detailed explanation, we refer to Section B.2.

Although Observation 2 suggests that minimal SISO-regions do not suffice when contemplating code duplication, we consider that the option of duplicating code remains a valuable strategy to decrease complexity and should not be neglected. Unfortunately, it seems impractical to cover all possible duplications because they can transform *any* subgraph into a SISO-region; thus, we believe it is necessary to establish a balance between the selection of nodes for duplication and the practicality of such endeavors. For example, when aiming to improve the readability of the decompiler output, one could limit duplications to linear code only, as analyzing complex structures repeatedly greatly reduces the analysis efficiency. This restriction would also imply that duplicating the entry node of a minimal SISO-region is never advantageous because it is either non-linear or consists of just two nodes, making it easily resolvable.

To identify all conditional regions where the single-successor/exit is duplicated, we can broaden our definition of SISO-regions to the following: We define an *extended SISO* region as a subgraph that can be converted into a SISO-region by duplicating linear code. As a consequence, we can now make the following observation:

Observation 3. When ignoring the duplication of non-linear code, considering only conditional regions of minimal extended SISO-regions still guarantees an optimal decompiler output.

Leveraging on Observation 3, we are now able to compute *all* extended SISO-regions for a given entry node within a feasible timeframe. For this, we drafted an approach in Section B.3 using the following idea: Essentially, we are able to identify the single-successor/exit of extended SISO-regions by utilizing graph-separators of size one and cutvertices in an auxiliary graph.

Overall, given the above established restriction of not duplicating non-linear code in considered conditional regions, it is possible to significantly reduce the number of subgraphs that future approaches need to consider to identify resolvable and conditional regions. Ultimately, future research should evaluate whether the restriction to allow only the duplication of linear code poses a considerable constraint with regards to their objective or not.

5.3. Resolving Order and Practicability

Finally, we discuss the issue of determining a resolving order while maintaining practicability, which will most likely necessitate compromises or trade-offs even when leveraging on our discussed approaches. Using our observations and methodologies above, an optimal method for determining the best resolving order could work as follows. For each node in post-order traversal, identify all conditional and resolvable regions and build a decision tree with branches representing choosing each possible region or none. This process is repeated for every node until the entire graph is translated, likely resulting in an exponential number of leaves. The set of leaf nodes includes all potential decompiler outputs, from which the optimal solution can be chosen based on a predefined metric or evaluation criterion. However, evaluating *all* regions and the consequently high number of resolving orders is clearly impractical and requires some restrictions.

Unfortunately, methods can practically only follow a fixed number of paths in the decision tree and must choose which conditional and resolvable regions to evaluate. Assuming that one would develop a practical approach based on our extended SISO-region computations, restricted to only duplicating linear code, we would propose traversing the nodes in post-order traversal. In such an approach, the evaluation of inner structures precedes that of outer structures, akin to current methods, to improve the output quality. Although our analysis already narrows down the regions to compute, it remains challenging to determine which regions to consider and how to choose among them.

For a given node h, we can initially check if a resolvable SISO-region exists by heuristically trying to translate the SISO-region from Observation 1 into a natural C-structure. In case no SISO-region with entry h exists, we recommend skipping the current node h and targeting a

SISO-region that includes node h. As previously shown in Figure 3d, premature code duplication that disrupts natural C-structures makes the possible identification of an extended SISO-region less appealing. However, for cyclic regions we suggest to *not* skip the current node to ensure that the cyclic region is translated as a loop; thus requiring a trade-off by handling multiple entries or exits similarly to existing research [2], [35], [73].

If the region proves to be resolvable, we advise choosing it, as it is unlikely to significantly increase the complexity compared to an optimal output, due to the absence of added complexity from using gotos, structural variables, or duplication. Moreover, it is unlikely to feasibly compute all possible translations, much less all scenarios where the single-entry or single-successor/exit is duplicated. However, considering the complexity of the translated high-level code, future methods might consider computing some additional conditional (sub)regions to determine whether the complexity can be reduced. For instance, if the single-successor is a return block, adding a duplicate into the region could simplify the output.

If the SISO-region with entry h is not resolvable, future approaches will probably need to make additional trade-offs, because determining *all* conditional regions within an extended SISO-region that has node h as an entry point seems impractical. Consequently, they likely need to heuristically select which regions to consider based on their objectives. Generally, we note that unlike the situation where a SISO-region is absent, no extended SISO-region with entry h can ever be part of a resolvable-region, ensuring that code duplication will not compromise any natural C-structure. In addition, to take advantage of the opportunity to duplicate the single successor/exit of a SISO-region, we recommend also examining extended SISO-regions with single-entry h in these cases. Nevertheless, it remains uncertain and subject to future research to decide whether smaller or larger extended SISO-regions yield better output or which regions are beneficial in general. Although, from a practical standpoint, considering smaller regions seems to be advantageous, this does not necessitate that omitting the selection of the smallest possible region renders the approach impractical.

In conclusion, we acknowledge that we cannot solve all issues of the four identified challenges and believe that future research will still rely on certain heuristics when developing a functional *and* practical control-flow structuring approach. First, determining whether a SISO-region is resolvable remains a challenge itself; to maintain practicality, it may be necessary to limit the size of the region or to impose a time limit before ceasing efforts to identify a natural C-structure. Second, the computation of extended SISO-regions should probably be constrained, potentially by setting a fixed upper limit or by deciding that calculating a linear number is sufficient. In this context, research may also need to decide which extended SISO-regions to compute and which options to favor when making a region resolvable. Third, possibilities, not options, to make a conditional region a resolvable region likely need to be restricted. Ultimately, we believe that the decision on the resolving order remains the most critical issue. While calculating a local metric *could* yield reasonably good results overall, a superior metric is needed beyond merely comparing (extended) SISO-regions.

6. Conclusion and Future Work

The recovery of high-level control-flow during decompilation remains a challenging research topic with numerous difficulties. Throughout this paper, we have identified the four main challenges that inevitably arise during control-flow structuring: defining resolvable subgraphs, handling non-resolvable subgraphs, determining a resolving order for (non-)resolvable regions, and ensuring practicability. We have highlighted that neglecting practicability can lead to a theoretically optimal control-flow recovery strategy, but this is neither helpful nor effective. Moreover, each challenge in isolation is already difficult: First, a broad definition of a resolvable region may hinder its efficient identification due to the lack of robust evidence that it is not resolvable. Second, there are numerous variations to transform a conditional region into a resolvable one. Third, the sheer number of resolvable and conditional regions makes it nearly impossible to locally determine a region for the best output.

Although not all challenges can be solved perfectly simultaneously, this does not mean that approaches should ignore them. Despite this, these challenges have not been adequately addressed by all previous researchers. By ignoring them, it is unclear whether all of their design decisions were made consciously with an understanding of the inherent limitations. Besides, we demonstrated that existing methods often fall short by not addressing a subset of these challenges, particularly the ability to handle non-resolvable subgraphs and their ordering or practicability, thereby yielding less than optimal results. Consequently, for authors and developers of future decompilation and control-flow structuring approaches, we propose that they:

1) clearly discuss how they address each challenge,
2) are aware of the available options and known limitations of each challenge,
3) and optimize with regard to their goals.

By explicitly addressing each challenge, we enable others to gain a better understanding of the distinctive benefits and drawbacks of novel approaches, while simultaneously enhancing the development of new methods or decompilers. In addition, considering these challenges significantly facilitates the comparison of approaches. We conclude that leveraging all possible options in alignment with the authors' goals can considerably boost the overall quality of a decompiler. Likewise, current decompilers should also reassess the existing options, such as duplication or gotos, that they previously overlooked.

Understanding the limitations of current methodologies leads us to explore the necessity for innovative solutions in the analysis of control-flow structuring. Although we demonstrated that computing only a subset of extended SISO-regions is sufficient, the reduction in the total number of regions remains inadequate. Consequently, the primary challenge for future approaches lies in developing suitable heuristics to obtain the best results while maintaining practicality. These heuristics will most likely depend on the selected objective or the chosen metric, although some might be universal. In summary, this thorough review of existing challenges, together with insights and proposals to identify significant regions in future approaches, provides a strong foundation for future research in the area of control-flow structuring.

References

[1] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[2] Frances E Allen and John Cocke. *Graph-theoretic constructs for program control flow analysis*. IBM Thomas J. Watson Research Center, 1972.

[3] Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael FP O'Boyle. Slade: A portable small language model decompiler for optimized assembly. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 67–80. IEEE, 2024.

[4] 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 *Proceedings of the USENIX Security Symposium*, 2024. For decompilation, we used the SAILR implementation integrated into angr 9.2.142; for some figures we used the SAILR implementation referenced in the paper.

[5] Marcus Botacin, Lucas Galante, Paulo de Geus, and André Grégio. Revenge is a dish served cold: Debug-oriented malware decompilation and reassembly. In *Proceedings of the 3rd Reversing and Offensive-oriented Trends Symposium*, pages 1–12, 2019.

[6] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 463–469. Springer, 2011.

[7] David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. Native x86 decompilation using {Semantics-Preserving} structural analysis and iterative {Control-Flow} structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 353–368, 2013. For decompilation, we used the Phoenix implementation integrated into angr 9.2.142.

[8] Kevin Burk, Fabio Pagani, Christopher Kruegel, and Giovanni Vigna. Decomperson: How humans decompile and what we can learn from it. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2765–2782, 2022.

[9] G. Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 international conference on technical debt*, pages 57–58, 2018.

[10] Ying Cao, Ruigang Liang, Kai Chen, and Peiwei Hu. Boosting neural networks to decompile optimized binaries. In *proceedings of the 38th annual computer security applications conference*, pages 508–518, 2022.

[11] Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. Evaluating the effectiveness of decompilers. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 491–502, 2024.

[12] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4327–4343, 2022.

[13] Cristina Cifuentes. A structuring algorithm for decompilation. In *Proceedings of the XIX Conferencia Latinoamericana de Inform atica*, pages 267–276. Citeseer, 1993.

[14] Cristina Cifuentes. *Reverse compilation techniques*. Citeseer, 1994.

[15] Cristina Cifuentes. Structuring decompiled graphs. In *International Conference on Compiler Construction*, pages 91–105. Springer, 1996.

[16] Cristina Cifuentes and K John Gough. A methodology for decompilation. In *Proceedings of the XIX Conferencia Latinoamericana de Inform atica*, pages 257–266, 1993.

[17] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to high-level language translation. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 228–237. IEEE, 1998.

[18] John Cocke. *Programming languages and their compilers: Preliminary notes*. New York University, 1969.

[19] John Cocke. Global common subexpression elimination. In *Proceedings of a symposium on Compiler optimization*, pages 20–24, 1970.

[20] John Cocke and Raymond Miller. Some analysis techniques for optimizing computer programs. In *Proceedings of the Second Intl. Conf. of Systems Science*, 1969.

[21] Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. rev. ng: A multi-architecture framework for reverse engineering and vulnerability discovery. In *2018 International Carnahan Conference on Security Technology (ICCST)*, pages 1–5. IEEE, 2018.

[22] Edsger W Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[23] Luke Dramko, Jeremy Lacomis, Edward J Schwartz, Bogdan Vasilescu, and Claire Le Goues. A taxonomy of c decompiler fidelity issues. In *33th USENIX Security Symposium (USENIX Security 24)*, 2024.

[24] Luke Dramko, Jeremy Lacomis, Pengcheng Yin, Ed Schwartz, Miltiadis Allamanis, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. Dire and its data: Neural decompiled variable renamings with respect to software class. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–34, 2023.

[25] Lukás Ďurfina, Jakub Křoustek, and Petr Zemek. Psybot malware: A step-by-step decompilation case study. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 449–456. IEEE, 2013.

[26] Lukáš Ďurfina, Jakub Křoustek, Petr Zemek, Dušan Kolář, Tomáš Hruška, Karel Masařík, and Alexander Meduna. Design of a retargetable decompiler for a static platform-independent malware analysis. In *Information Security and Assurance: International Conference, ISA 2011, Brno, Czech Republic, August 15-17, 2011. Proceedings*, pages 72–86. Springer, 2011. For decompilationm, we used RetDec Version v5.0 (53e55b4b).

[27] Steffen Enders, Eva-Maria C. Behner, Niklas Bergmann, Mariia Rybalka, Elmar Padilla, Er Xue Hui, Henry Low, and Nicholas Sim. dewolf: Improving decompilation by leveraging user surveys. *NDSS Workshop on Binary Analysis Research*, 2023.

[28] Steffen Enders, Mariia Rybalka, and Elmar Padilla. Pidarci: Using assembly instruction patterns to identify, annotate, and revert compiler idioms. In *2021 18th International Conference on Privacy, Security and Trust (PST)*, pages 1–7. IEEE, 2021.

[29] Felix Engel, Rainer Leupers, Gerd Ascheid, Max Ferger, and Marcel Beemster. Enhanced structural analysis for c code reconstruction from ir code. In *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*, pages 21–27, 2011.

[30] Haeun Eom, Dohee Kim, Sori Lim, Hyungjoon Koo, and Sungjae Hwang. R2i: A relative readability metric for decompiled code. *Proceedings of the ACM on Software Engineering*, 1(FSE):383–405, 2024.

[31] Ana M Erosa and Laurie J Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*, pages 229–240. IEEE, 1994.

[32] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. Smartdec: approaching c++ decompilation. In *2011 18th Working Conference on Reverse Engineering*, pages 347–356. IEEE, 2011.

[33] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 32, 2019.

[34] Ilfak Guilfanov. Decompilers and beyond. *Black Hat USA*, 9:46, 2008.

[35] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A comb for decompiled c code. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 637–651, 2020. Utilized decompiler version 943ae3b.

[36] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680, 2018.

[37] Matthew S Hecht. *Flow analysis of computer programs*. Elsevier Science Inc., 1977.

[38] Hex-Rays SA. Hex-rays decompiler. https://www.hex-rays.com/products/decompiler/, 2024. IDA Version 8.3.

[39] Peiwei Hu, Ruigang Liang, and Kai Chen. Degpt: Optimizing decompiler output with llm. In *Proceedings 2024 Network and Distributed System Security Symposium (2024). https://api. semanticscholar. org/CorpusID*, volume 267622140, 2024.

[40] Alan Jaffe, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of the 26th Conference on Program Comprehension*, pages 20–30, 2018.

[41] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 171–185, 1994.

[42] Deborah S Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*, pages 346–356. IEEE, 2018.

[43] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019.

[44] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. 2011.

[45] Ruigang Liang, Ying Cao, Peiwei Hu, and Kai Chen. Neutron: an attention-based neural decompiler. *Cybersecurity*, 4:1–13, 2021.

[46] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*, pages 1–1, 2010.

[47] Ziyi Lin, Jinku Li, Bowen Li, Haoyu Ma, Debin Gao, and Jianfeng Ma. Typesqueezer: When static recovery of function signatures for binary executables meets dynamic analysis. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2725–2739, 2023.

[48] Zhibo Liu and Shuai Wang. How far we have come: Testing decompilation correctness of c decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 475–487, 2020.

[49] Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. {RE-Mind}: a first look inside the mind of a reverse engineer. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2727–2745, 2022.

[50] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *European Symposium on Programming*, pages 208–223. Springer, 1999.

[51] National Security Agency. Ghidra software reverse engineering framework. https://ghidra-sre.org/, 2024. Version 11.3.1.

[52] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, et al. ”len or index or count, anything but v1”: Predicting variable names in decompilation output with transfer learning. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4069–4087. IEEE, 2024.

[53] Si Pan and R. Geoff Dromey. A formal basis for removing goto statements. *The Computer Journal*, 39(3):203–214, 1996.

[54] Daniel Plohmann. SMDA: A minimalist recursive disassembler library. https://github.com/danielplohmann/smda, 2025.

[55] Daniel Plohmann, Martin Clauss, Steffen Enders, and Elmar Padilla. Malpedia: a collaborative effort to inventorize the malware landscape. *The Journal on Cybercrime and Digital Investigations*, 3(1):1–19, 2017.

[56] Radare2 Project. Radare2. http://www.radare.org/r/, 2016. Versions r2-5.9.9, r2pm-5.9.9, r2dec-dd344249.

[57] Lyle Ramshaw. Eliminating go to's while preserving program structure. *Journal of the ACM (JACM)*, 35(4):893–920, 1988.

[58] Md Omar Faruk Rokon, Risul Islam, Ahmad Darki, Evangelos E Papalexakis, and Michalis Faloutsos. {SourceFinder}: Finding malware {Source-Code} from publicly available repositories in {GitHub}. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 149–163, 2020.

[59] Barry K Rosen. High-level data flow analysis. *Communications of the ACM*, 20(10):712–724, 1977.

[60] Eric Schechter. *Handbook of Analysis and its Foundations*. Academic Press, 1996.

[61] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *Workshop on Binary Analysis Research (BAR)*, 2018.

[62] Micha Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5(3-4):141–153, 1980.

[63] Mary Shaw. Abstraction techniques in modern programming languages. *IEEE software*, 1(04):10–26, 1984.

[64] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.

[65] Vector 35 Inc. Binary ninja. https://binary.ninja/, 2024. Version 4.2.6455 (02c8da1e).

[66] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. Sound c code decompilation for a subset of x86-64 binaries. In *International Conference on Software Engineering and Formal Methods*, pages 247–264. Springer, 2020.

[67] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse {Engineers'} processes. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1875–1892, 2020.

[68] Rusty Wagner. Restructuring the binary ninja decompiler. https://binary.ninja/2024/06/19/restructuring-the-decompiler.html, June 19 2024. Accessed: 2024-06-28.

[69] M. Howard Williams and G Chen. Restructuring pascal programs containing goto statements. *The Computer Journal*, 28(2):134–137, 1985.

[70] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4554–4568, 2024.

[71] Khaled Yakdan. *A Human-Centric Approach For Binary Code Decompilation*. PhD thesis, Bonn University, 2018.

[72] Khaled Yakdan, Sebastian Eschweiler, and Elmar Gerhards-Padilla. Recompile: A decompilation framework for static analysis of binaries. In *2013 8th International Conference on Malicious and Unwanted Software:" The Americas"(MALWARE)*. IEEE, 2013.

[73] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*. Citeseer, 2015. For decompilation, we used the DREAM implementation integrated into angr 9.2.142; for some figures we used dewolf with a respective configuration, because both follow the same methodology.

[74] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 813–832. IEEE, 2021.

[75] Chang Zhu, Ziyang Li, Anton Xue, Ati Priya Bajaj, Wil Gibbs, Yibo Liu, Rajeev Alur, Tiffany Bao, Hanjun Dai, Adam Doupé, et al. {TYGR}: Type inference on stripped binaries using graph neural networks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4283–4300, 2024.

[76] Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave Jing Tian. {D-Helix}: A generic decompiler testing framework using symbolic differentiation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 397–414, 2024.

Appendix A. Data Availability

With the publication of this paper, we also publish all related artifacts[1]. Primarily, this includes all binaries (and their source code) that we used to generate the examples in Sections 4.1 to 4.4. Because we cannot guarantee that all considered decompilers will be available in the future, we also provide the unmodified decompiler outputs used for the figures. Regarding the referenced Malpedia samples, which we utilized for argumentation purposes and assessment evaluation, we refer to the Malpedia website[2]. Finally, we publish all scripts that we used to decompile samples and generate the assessment results.

Appendix B. Theoretical considerations

B.1. Observation 1: Defining a Total Preorder

First, we show the correctness of Observation 1, i.e., that there exists a resolvable region that does not decrease the readability of the output more than choosing any other resolvable region with the same single-entry. Therefore, we define a total preorder[3] on all resolvable regions with the same single-entry. Any maximal element of the total preorder is a resolvable region for the given entry that leads to an output that does not decrease the output quality compared to choosing any other resolvable region. Thus, each approach only has to compute one of these regions.

Let R_1 and R_2 be two resolvable regions for a given entry node. To define a total preorder on the set of all resolvable regions with the same single-entry, we have to set these two sets in a relation. We assume for simplicity that they are both single-successor regions; otherwise, we apply the transformation of Figure 2. If neither $R_1 \subseteq R_2$ nor $R_2 \subseteq R_1$, then $R_2 \setminus R_1$ and $R_1 \setminus R_2$ are resolvable regions: The set $R_1 \setminus R_2$ and the set $R_2 \setminus R_1$ is dominated by the entry since both sets are contained in a resolvable region with this entry. Consequently, both sets are single-entry regions without any successors, where the entry is the successor of R_1 resp. R_2, that are part of a resolvable region, implying that they are resolvable. Since we consider inner-structures before outer-structures, these two regions should have been translated first. Thus, the sets $R_2 \setminus R_1$ and $R_1 \setminus R_2$ consist of only one node.

In general, we prefer R_1 over R_2, i.e., $R_2 \leq R_1$, if the complexity of $R_2 \setminus R_1$ is higher than the complexity of $R_1 \setminus R_2$ according to the chosen complexity measurement. By symmetry, $R_1 \leq R_2$ if the complexity of $R_1 \setminus R_2$ is higher than the complexity of $R_2 \setminus R_1$. Choosing such an order helps to reduce the complexity inside the structure, especially if we consider loops. Nevertheless, for *cyclic* regions we have one exception: If region R_1 can be translated into a while-loop or do-while loop, then we prefer R_1 over R_2, i.e., $R_2 \leq R_1$ to prefer loops with a condition over endless-loops. By symmetry, if the region R_2 can be translated into a (do-)while- loop, then $R_1 \leq R_2$. Additionally, to avoid destroying natural C-structures when resolving *acyclic* regions, resolving $R_1 \cup R_2$ is preferable over R_1 and R_2, i.e., $R_1, R_2 \leq R_1 \cup R_2$. Note that the region $R_1 \cup R_2$ does not have successors or exits.

Next, if $R_1 \subseteq R_2$ and these sets are not ordered by the first rule, it is always beneficial to prefer R_1 over R_2, since R_1 is an inner-structure of R_2, i.e., $R_2 \leq R_1$. Besides, in this scenario we do not destroy any structure by resolving R_1 (inner structure) before R_2 (outer structure). Similarly, if $R_2 \subseteq R_1$ and if these two sets are not already ordered, then $R_1 \leq R_2$ holds by symmetry. In total, this defines a total preorder of all resolvable regions and additionally demonstrates that computing one is enough; which poses a novel perspective compared to existing approaches.

B.2. Observation 2 + 3: Minimal SISO-Regions

Now, we argue for the correctness of Observation 2, i.e., that considering only minimal SISO-regions yields optimal output, given that an optimal solution does not duplicate the single-entry or the single-successor/exit. As mentioned before, given a SISO-region, the flow can only enter through the single-entry and can only leave through the single-successor/exit. Thus, if neither of these nodes is duplicated, the flow in the transformed resolvable (sub)graph also must enter through the single-entry and leave through the single-successor/exit. Consequently, the resulting output always contains a C-structure that matches this SISO-region for all resolving orders. Hence, resolving this C-structure at once instead of first considering smaller subgraphs does not decrease the output quality.

While Observations 2 and 3 assert that considering (extended) SISO-regions is sufficient to achieve an optimal output under specific restrictions, two scenarios may benefit from the inclusion of additional regions in future research: First, if the CFG is irreducible, it contains a loop with a back-edge whose source is not dominated by the sink. Therefore, such a loop will never be included in an (extended) SISO-region where the entry is any of these loop nodes because the loop has multiple entries. It remains a viable approach to consider such a loop as part of a larger (extended) SISO-region, which potentially results in the omission of the loop. However, resolving loops is a crucial step in translating the CFG into high-level code. Thus, we recommend that approaches transform these loops into single-entry loops before attempting to identify (extended) SISO-regions. There are numerous methodologies that facilitate the transformation of irreducible graphs into reducible graphs [2], [35], [37], [73].

Second, there may be loops with a single-entry but no associated (extended) SISO-region with the same single entry, also called multiple-exit loops. Here, it is impossible to add nodes to the loop to obtain a single-successor/exit cyclic region. Similarly to multiple-entries, such a loop-region can be considered if it is contained in a larger (extended) SISO-region. However, control-flow structuring algorithms should probably translate every loop. Consequently, new approaches should also identify methods to make these loops resolvable. More precisely, they should consider conditional regions where the entry aligns with the loop-entry to ensure the resolving of the loop. In Section B.4, we present an idea for defining such regions.

[1] https://github.com/fkie-cad/control-flow-structuring-artifacts

[2] https://malpedia.caad.fkie.fraunhofer.de

[3] A *preorder* on a set is a relation $\leq$ that is transitive and reflexive. A *total preorder* is a preorder in which two elements are comparable [60].

B.3. Extended SISO-regions

In this section of the appendix, we describe our approach to compute and construct extended SISO-regions. To find all extended SISO-regions, we construct an auxiliary graph H and use *vertex-separators* and *cutvertices*. The idea for this is that the vertex-separator of size one resp. the cutvertex is the single-successor/exit of the extended SISO-region. We start by defining these two terms:

Definition 1. Let $G = (V, E)$ be an undirected graph and let s, t be two distinct vertices in G.

1) A s, t*-vertex separator* is a vertex set $X \subseteq V \setminus \{s, t\}$ s.t. there is no s, t-path in $G - X$.
2) A *cutvertex* is a node $w \in V$, s.t. $G - \{w\}$ has more connected components than G.
3) For a vertex set X, the set of reachable nodes from s in $G - X$ is denoted by $R_G(s, X)$.

We note that each s, t-vertex separator of size one in G is also a cutvertex in G, but not every cutvertex in G is a s, t-separator. To be able to construct the auxiliary graph and to construct extended SISO-regions we have to define a few node sets. Some of these node-sets depend on a node h, which is in these cases the single-entry of the (extended) SISO-region. In case the node h has an out-degree one, node h together with its unique successor is a SISO-region, and also a resolvable region: First, the code represented in h is executed, followed by the code contained in the unique successor. In these cases, each approach should restructure these two nodes, and the construction of these node sets is not necessary, i.e., we only suggest to compute these sets if node h has at least two successors. In total, we define the following six node sets:

D The set of all nodes in the CFG G that can be duplicated. Given our established restriction, these are all nodes containing linear code, thus having an out-degree of at most one.

dom(h) The set of all nodes dominated by h.

dom$^+$(h) The set of all nodes dominated by h together with the duplicatable nodes reachable from h in $G[\mathrm{dom(h)} \cup \mathrm{D}]$.

N The set of all successors of $\mathrm{dom}^+(\mathrm{h})$.

D$_h$ All duplicatable nodes in $\mathrm{dom}^+(\mathrm{h})$.

dom$^-$(h) The set of all nodes of $\mathrm{dom}^+(\mathrm{h})$ without D_h, i.e., $\mathrm{dom(h)} \setminus \mathrm{D}_h$.

Given these sets, every extended SISO-region with entry h is contained in $\mathrm{dom}^+(\mathrm{h})$, because all other nodes have predecessors that are not dominated by h and can not be duplicated. Thus, by adding one of these nodes, we would not get a single-entry region. Consequently, an extended SISO-region with entry h never contains any node of N, since these are the successors of $\mathrm{dom}^+(\mathrm{h})$.

Next, we construct the undirected auxiliary graph H, with its nodes consisting of the sets $\mathrm{dom}^-(\mathrm{h})$ and N together with an additional node t. The set of edges is derived from the CFG in the following manner: For every directed edge (x, y) in the CFG with $x \in \mathrm{dom}^-(\mathrm{h})$ and $y \in \mathrm{dom}^-(\mathrm{h}) \cup \mathrm{N}$, we add the undirected edge $\{x, y\}$ to the auxiliary graph. Furthermore, for each node $x \in \mathrm{N}$, we add the undirected edge $\{x, t\}$ to the auxiliary graph. In summary, H is the undirected graph with node set $V(H) = \mathrm{dom}^-(\mathrm{h}) \cup \mathrm{N} \cup \{t\}$ and edge set $E(H) = \{\{x, y\} \mid (x, y) \in E(G), x \in \mathrm{dom}^-(\mathrm{h}), y \in \mathrm{dom}^-(\mathrm{h}) \cup \mathrm{N}\} \cup \{\{n, t\} \mid n \in \mathrm{N}\}$. Figure 8 illustrates the construction of this auxiliary graph H.

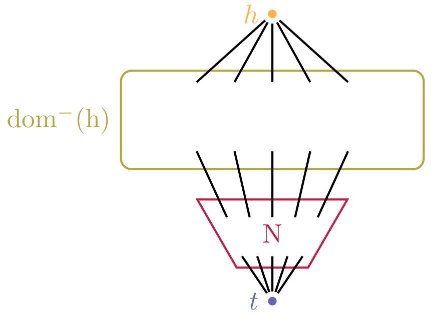


Figure 8: Illustration of the auxiliary graph H used for the computation of the (extended) SISO-regions.

We recall that every extended SISO-region is contained in the set $\mathrm{dom}^+(\mathrm{h}) = \mathrm{dom}^-(\mathrm{h}) \cup \mathrm{D}_h$. Furthermore, given an extended SISO-region R, we can modify the CFG by duplicating all nodes contained in the set $\mathrm{D}^M = \{l \in R \cap \mathrm{D} \mid N_G^-(l) \nsubseteq R \setminus \mathrm{D}^M\} \subseteq D$ to obtain a SISO-region. More precisely, these are all duplicatable nodes that either have a predecessor outside the region or a predecessor that needs to be duplicated, i.e., is contained in the set D^M. Duplicating these nodes ensures the single-entry property while duplicating a minimum number of nodes. In the remainder of this section, we will show that every h, t-vertex separator of size one, resp. every cutvertex corresponds to an extended SISO-region, where node t is the additional node in the constructed auxiliary graph H and h is the single-entry. Additionally, we will argue that our approach can find all extended SISO-regions based on the following observations:

1) The only possible extended SISO-region without any successor is $\mathrm{dom}^+(\mathrm{h})$ and it exists if $\mathrm{N} = \emptyset$.
2) For every extended SISO-region R with single-entry h and single-successor/exit w, one of the following holds[4]:
   (i) The node w is an h, t-vertex separator of size at most one or a cutvertex in H.
   (ii) The single-entry h is identical to the single exit w, that is, $h = w$.
   (iii) The set $\mathrm{dom}^-(\mathrm{h})$ is contained in R and $\mathrm{N} = \emptyset$.

Overall, these two statements imply that we can find the single-successor/exit of every extended SISO-region by identifying all graph-separators of size at most one as well as all cutvertices together with the regions containing $\mathrm{dom}^-(\mathrm{h})$ or having h as single-exit. Next, we describe how we construct the extended SISO-regions in each of these cases: The construction of all extended SISO-regions is divided into multiple cases, depending on the existence of an h, t-path in H and some properties of the entry node h. In total, we describe six ways how to construct extended SISO-regions, although Constructions 2 and 3 are only for acyclic SISO-regions and Construction 6 is the equivalent construction for cyclic SISO-regions. As

[4] Proof idea: If (ii) or (iii) hold, we have nothing to prove. Otherwise, we have $w \in \mathrm{dom}^-(\mathrm{h}) \cup \mathrm{N}$. Now, if w is neither an h, t-vertex separator of size 1 nor a cutvertex in H, then one can show that R is not a single-successor/exit region or contains an inner-loop structure.

already mentioned, for cyclic regions, the SISO-regions should contain a loop of the CFG. In the following, we will denote the loop whose SISO-region we want to compute as L. For each of the constructions, we include an illustration in Figure 9 and basic ideas on how we proved that they hold. First, we consider the basic construction in the case where an h, t-path in H exists:

Construction 1. Assume that there is an h, t-path in H. Let $w \in V(H)$ be an h, t-vertex separator of size one in H, let $R = R_H(h, \{w\})$ and let $\mathcal{C}$ be the set of connected components of $H - w$ containing neither node h nor node t.

(i) If $N_G^+(w) \cap R = \emptyset$ then $R_G(h, \{w\})$ is an extended SISO-region.[5]

(ii) If node $w \notin N$ then $R_G(h, X)$ is an extended SISO-region for all connected components $\mathcal{C}' \subseteq \mathcal{C}$ and all node sets $\mathrm{DS}_w \subseteq N_G^+(w) \cap \mathrm{D}_h$ with node set $X = N_G^+(w) \setminus (\bigcup \mathcal{C}' \cup \mathrm{DS}_w)$.[6]

Next, we consider the case that node h is a cutvertex in H. This construction is independent of the existence or absence of an h, t-path in H, and leads to the second and third constructions:

Construction 2. Assume node h is a cutvertex in H. Let $\mathcal{C}$ be the set of connected components in $H - h$ not containing t. For all $\mathcal{C}' \subseteq \mathcal{C}$ and for all $\mathrm{DS}_h \subseteq N_G^+(h) \cap \mathrm{D}_h$ it holds that $R_G(h, X)$ is an extended SISO-region where $X = N_G^+(h) \setminus (\bigcup \mathcal{C}' \cup \mathrm{DS}_h)$.[7]

However, node h can also be the single-exit of an extended SISO-region without being a cutvertex. This extended SISO-region consists of node h together with duplicatable nodes of the set D_h, thus requiring the following construction, except when h is a cutvertex:

Construction 3. Assume node h has duplicatable neighbors, i.e., $N_G^+(h) \cap \mathrm{D}_h \neq \emptyset$. For all $\mathrm{DS}_h \subseteq N_G^+(h) \cap \mathrm{D}_h$ it holds that $R_G(h, N_G^+(h) \setminus \mathrm{DS}_h)$ is an extended SISO-region.[8]

The described constructions cover the cases where there is an h, t-path in H and with node h being the single-exit. Thus, the constructions for the cases without an h, t-path in H remain. In these cases, nodes h and t can already be considered separated; hence, we are only interested in cutvertices or regions that result from the fact that there is no h, t-path. We start with the construction for the latter:

Construction 4. Assume that there is no h, t-path in H.

(i) $\mathrm{dom}^+(\mathrm{h})$ is an extended SISO-region.[9]

(ii) $\mathrm{dom}^+(\mathrm{h}) \setminus \mathrm{dom}(\mathrm{w})$ is an extended SISO-region for all nodes $w \in \mathrm{D}_h$.[10]

(iii) Let $w \in \mathrm{dom}^-(\mathrm{h})$ s.t. $\mathrm{dom}(\mathrm{w}) \cap \mathrm{D}_h \neq \emptyset$[11]

a) $R_G(h, X)$ is an extended SISO-region for all non-empty sets $X \subseteq \mathrm{dom}(\mathrm{w}) \cap \mathrm{D}_h \cap N_G^+(w)$.

b) If $N_G^+(w) \subseteq \mathrm{D}_h$ then $\mathrm{dom}^+(\mathrm{h}) \setminus \mathrm{dom}(\mathrm{w})$ is an extended SISO-region.

Next, we show how to construct a region for a given cutvertex. Since Constructions 2 and 3 also hold when there is no h, t-path in H, we only have to consider cutvertices that are different from h for the final construction.

Construction 5. Assume that there is no h, t-path in H. Let $w \in V(H)$ be a cutvertex in H, let $R = R_H(h, \{w\})$ and let $\mathcal{C}$ be the set of connected components in $H - w$ containing neither h nor t.[12]

(i) If $N_G^+(w) \cap R = \emptyset$ then $R_G(h, \{w\})$ is an extended SISO-rebion.

(ii) $R_G(h, X)$ is an extended SISO for all connected components $\mathcal{C}' \subset \mathcal{C}$ and for all node sets $\mathrm{DS}_w \subseteq N_G^+(w) \cap \mathrm{D}_h$ with $X = N_G^+(w) \setminus (\bigcup \mathcal{C}' \cup \mathrm{DS}_w)$.

To ensure that constructions Constructions 1, 4 and 5 are cyclic extended SISO-regions containing the initial loop L, we have to show that loop L is contained in every constructed region: Primarily, no node in L is in D_h, because each of them has a successor in the initial loop region L. This implies that none of them is a leaf. Additionally, if they have a duplicatable leaf as a descendant, they have an out-degree of at least two and are therefore not duplicatable. Consequently, the initial loop region is contained in $\mathrm{dom}^-(\mathrm{h})$ and every node in the initial loop region has a successor in $\mathrm{dom}^-(\mathrm{h})$, implying that L is contained in each region constructed by Construction 4. Since the initial loop region has a vertex connectivity of at least two, it holds for the regions constructed via Constructions 1 and 5 that $\mathrm{L} \setminus \{w\}$ is contained in R, implying that L is in the constructed region because either $w \notin \mathrm{L}$ or $N_G^+(w) \cap R \neq \emptyset$.

Next, we modify the construction for the case that h is also the exit node of the cyclic region:

Construction 6. Assume that the node h is a cutvertex in H or $N_G^+(h) \cap D_h \neq \emptyset$. Let $\mathcal{C}$ be the set of connected components in $H - h$ that neither contains t nor any vertex of L, and let $\mathcal{C}_L$ be the set of connected components that contain the vertices of $\mathrm{L} \setminus \{h\}$.

If any connected component in $\mathcal{C}_L$ also contains node t, then there exists no extended cyclic SISO-region with exit node h.

Otherwise, for all connected components $\mathcal{C}' \subseteq \mathcal{C}$ and for all sets $\mathrm{DS}_h \subseteq N_G^+(h) \cap \mathrm{D}_h$, it holds that $R_G(h, N_G^+(h) \setminus (\bigcup(\mathcal{C}_L \cup \mathcal{C}') \cup \mathrm{DS}_h)$ is an extended SISO-region, i.e., the extended SISO-region contains $\bigcup \mathcal{C}' \cup \mathrm{DS}_h \cup \bigcup \mathcal{C}_L$, the node h, and some duplicatable nodes to ensure that h is the single-entry and single-exit node of the region.[13]

Overall, by introducing Constructions 1 to 6, we provide an approach to identifying all extended SISO-regions for a given entry by computing h, t-separators of size one and cutvertices.

[5] Proof idea: It is implied that w is the single-successor because all nodes that are reachable without w are contained in the region. It remains to show that the region has a single-entry after duplicating D^M. If this is not the case, then w is not a s, t-separator of in H.

[6] Proof idea: Since X contains only successors of w, this node is the single-exit. The proof that the region has a single-entry after duplicating D^M is similar to Construction 1 (i).

[7] Proof Idea: Similar to the idea described for Construction 1 (ii).

[8] Proof idea: The region consists only of h and duplicatable nodes in D_h, therefore, we can duplicate all necessary nodes in D_h to obtain a single-entry, single-exit region where the entry and exit node is h.

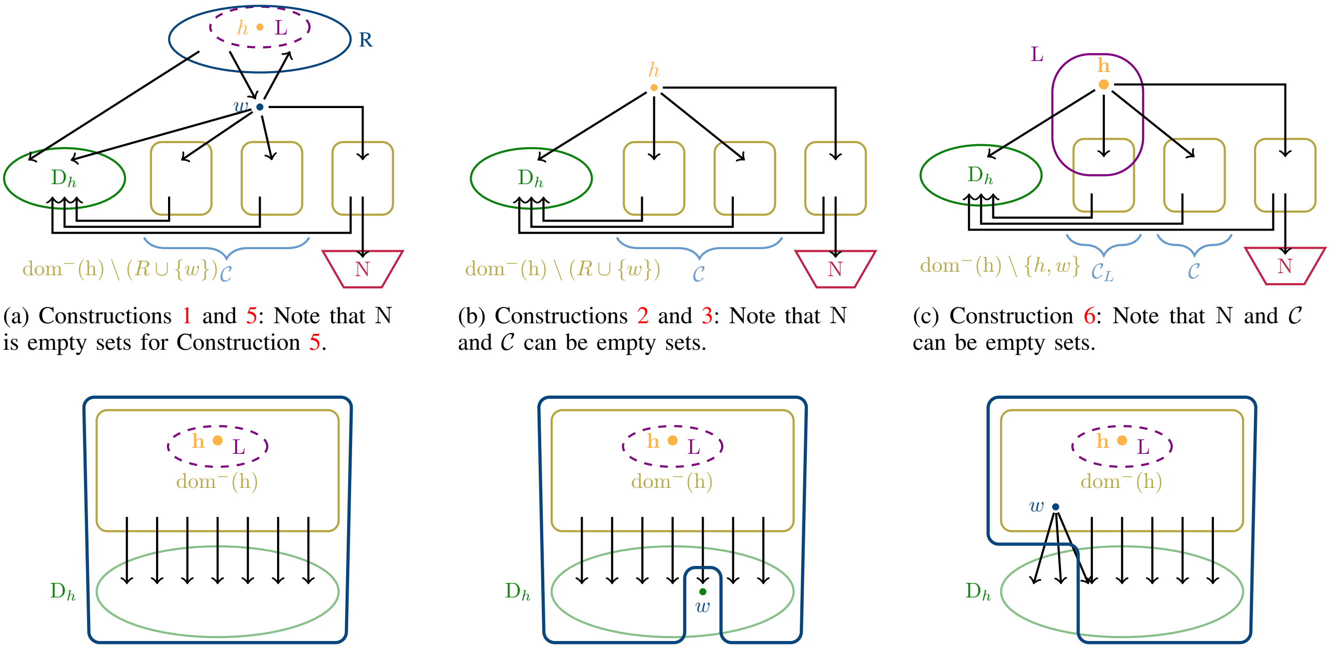
[9] Proof idea: Since $\mathrm{N} = \emptyset$, the region has no successor and duplicating nodes in D_h is sufficcent to obtain a single-entry.

[10] Proof idea: $\mathrm{dom}^+(\mathrm{h})$ is a SISO-region without an successor. Removing $\mathrm{dom}(\mathrm{w})$ implies that w is the single-successor, since all nodes in $\mathrm{dom}(\mathrm{w})$ are only reachable over w.

[11] Proof idea: Similar to the idea described for Construction 4 (ii).

[12] Proof idea: Similar to the idea described for Construction 1.

[13] Proof idea: Similar to the idea described for Constructions 2 and 3.



(a) Constructions 1 and 5: Note that N is empty sets for Construction 5.

(b) Constructions 2 and 3: Note that N and $\mathcal{C}$ can be empty sets.

(c) Construction 6: Note that N and $\mathcal{C}$ can be empty sets.

(d) Construction 4 (i): the only region without any successor/exit.

(e) Construction 4 (ii): regions with the successor in D$_h$.

(f) Construction 4 (iii): regions with the successor/exit in dom$^-$(h).

Figure 9: Illustration of all six constructions of extended SISO-regions, where L is a loop in the CFG whose extended SISO-region we want to compute when considering cyclic regions, otherwise L does not exist.

Runtime Considerations. Finally, we discuss whether the approach introduced in this section is practical applicable or not. First, since we are only interested in vertex-separators of size at most one and cutvertices, we can find these vertices in quadratic time. The worst-case algorithm would check for each node v in $V(H)$ whether node t is still reachable from h if v is removed from H, respectively H without node v has more connected components than H. Both properties can be easily verified by a modified breadth-first-search algorithm.

Furthermore, although our constructions seem to be quite complicated at first, we only compute a *linear* number of extended SISO-regions in total: Since the out-degree is constant c, and most nodes have out-degree at most two, each of the Constructions 1 to 3, 5 and 6 generates at most 2^c different SISO-regions. Furthermore, Construction 4 generates the region dom$^+$(h), one region for each node in D$_h$, and at most $2^c - 1$ regions for each node in dom$^-$(h), which is still linear in the number of nodes. Additionally, computing the sets $R_h(h, \_)$ can be done in linear time using a breadth-first-search algorithm. Nevertheless, in the case the single-exit has more than two successors, future approaches may want to reduce the number of regions they compute.

B.4. On Multiple-Exit Regions

As aforementioned, in case of cyclic regions, we always want to consider a subgraph where the single-entry is the loop-entry. Thus, in case there exists no extended SISO-region whose entry is the loop-entry, we suggest the computation of a subgraph with a minimum number of successors. By minimizing the number of successors, a minimal number of edges must be handled to obtain an extended SISO-region which hopefully reduces the introduced complexity. The single-entry of this subgraph should still be the loop-entry. Again, we argue for allowing the duplication of linear code, i.e., computing the minimal number of successors after potentially duplicating linear code. Here, we need another auxiliary graph H^* which, in contrast to the previous auxiliary graph H, is directed. However, the set of nodes is the same, except that we contract L into a single node. Since we allow the duplication of linear code, these remaining successors cannot be eliminated by code duplication; hence, we only have to consider the other available options to translate this region into a resolvable one.

For the construction of graph H^*, we first have to define the set of *ancestors* A(x), as the set of all nodes reaching node x, i.e., A$(x) = \{v \in$ dom$^-$(h) $\mid \exists v, x$-path$\}$. We need this set to ensure that all predecessors of a region node are also contained in the region; otherwise the region would not have a single entry. Now, we can define the graph H^*: The node set $V(H^*)$ consists of all nodes of the sets dom$^-$(h)$\setminus$L, N, one node s for the initial loop region L, and an additional sink node t. Similar to the graph H, we add each directed edge $(x, y) \in E(G)$ of the CFG G to the auxiliary graph H^* if the source x is in dom$^-$(h)$\setminus$L and the sink y is in dom$^-$(h)$\setminus$L$\cup$N, and a directed edge from each node in N to t. Furthermore, we add an edge between node s and a node y in dom$^-$(h)$\setminus$L$\cup N$ if there exists a node $x \in$ L such that $(x, y) \in E(G)$. Finally, we add the edge set $\hat{E}$ to graph H^*, consisting of the edges (x, y) with $x \in$ dom$^-$(h) $\setminus$ L and $y \in N_G^+(\mathrm{A}(x))$. By the construction of graph H^*, it holds that there exists no edge from N to dom$^-$(h) in H^* and that node s has no predecessors in H^*. To find a region with a minimum number of successors, we compute the closest minimum s, t-vertex separator in graph H^*.

Definition 2. Let $G = (V, E)$ be a directed graph and let s, t be two distinct vertices in G.

1) A s, t-*vertex separator* is a vertex set $X \subseteq V \setminus \{s, t\}$ s.t. there is no s, t-path in $G - X$. Note, there can still be t, s-path in $G - X$.
2) For a vertex set X, the set of reachable nodes from node s in $G - X$ is denoted by $R_G(s, X)$.

We chose the closest separator to obtain a loop-region that does contain as few additional nodes as possible. Basically, only the nodes of the loop are contained in a cycle, so each node we add is somehow not part of a cycle. Now, given a minimum s, t-separator of size ℓ in graph H^*, we can construct the following single entry region with ℓ successors:

Construction 7. Let set X be a closest minimum s, t-separator in H^*. $R = R_G(h, X)$ is a single-entry region with $|X|$ successors containing L after potentially duplicating some linear-code.

Here we suggest only computing one region, to not artificially inflate the loop-region. Afterwards, we only have to duplicate the nodes in $R \cap \mathrm{D}$ having a predecessor outside the region or whose unique successor must be duplicated. Of course, approaches can also consider subgraphs of this region and translate these into C-structures.