Ahoy SAILR! There is No Need to DREAM of C: A Compiler-Aware Structuring Algorithm for Binary Decompilation

Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang, *Arizona State University*

https://www.usenix.org/conference/usenixsecurity24/presentation/basque

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Ahoy SAILR! There is No Need to DREAM of C: A Compiler-Aware Structuring Algorithm for Binary Decompilation

Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao,
Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, Ruoyu Wang
Arizona State University
{zbasque,atipriya,wfgibbs,judeo,derronm,tbao,doupe,yans,fishw}@asu.edu

Abstract

Contrary to prevailing wisdom, we argue that the measure of binary decompiler success is not to eliminate all gotos or reduce the complexity of the decompiled code but to get as close as possible to the original source code. Many gotos exist in the original source code (the Linux kernel version 6.1 contains 3,754) and, therefore, should be preserved during decompilation, and only *spurious gotos* should be removed.

Fundamentally, decompilers insert spurious gotos in decompilation because structuring algorithms fail to recover C-style structures from binary code. Through a quantitative study, we find that the root cause of spurious gotos is compiler-induced optimizations that occur at all optimization levels (17% in non-optimized compilation). Therefore, we believe that to achieve high-quality decompilation, decompilers must be *compiler-aware* to mirror (and remove) the goto-inducing optimizations.

In this paper, we present a novel structuring algorithm called SAILR that mirrors the compilation pipeline of GCC and precisely inverts goto-inducing transformations. We build an open-source decompiler on `angr` (the ANGR DECOMPILER) and implement SAILR as well as otherwise-unavailable prior work (Phoenix, DREAM, and rev.ng's Combing) and evaluate them, using a new metric of how close the decompiled code structure is to the original source code, showing that SAILR markedly improves on prior work. In addition, we find that SAILR performs well on binaries compiled with non-GCC compilers, which suggests that compilers similarly implement goto-inducing transformations.

1 Introduction

Compiled programs called binaries power the devices that drive modern society. However, many of these binaries are shipped without accompanying source code, making their analysis arduous. To reduce this burden, decompilation techniques aim to recover source code from binary code. Security analysts use decompilation in reverse engineering [11, 31], malware analysis [19, 20, 33, 43], and vulnerability discovery and mitigation [32, 36].

Many applications of decompilation require *high-quality* decompiled source code. One important criterion of high-quality source code is meaningful control flow structures, such as `if-else` statements, `while` loops, and `do-while` loops in the C language. Unfortunately, the semantics of these control flow structures are lost during compilation, replaced by simple binary-level control flow transfers such as `jmp` instructions.

Decompilers leverage control flow *structuring* algorithms that analyze low-level constructs in the compiled binary and attempt to recover the high-level control flow. If compilers simply translated C code to assembly and pieced the assembly together with `jmp` instructions, the resulting code would be easily *structurable*, and decompilers would be able to produce high-quality decompiled source code. However, modern compilers optimize and distort code structures during compilation, making the result *unstructurable* and preventing current structuring algorithms from recovering the high-level control flow structure.

Structuring failures manifest as *goto* statements in the decompilation, stitching together portions of code where the decompiler failed to recover reasonable high-level structures [44]. The fact that it is specifically a goto that results from such failures has caused some semantic confusion between the Software Engineering and Decompilation research communities. In Software Engineering, *gotos are considered harmful*, with Dijkstra famously stating that "the quality of programmers is a decreasing function of the density of goto statements in the programs they produce" [18]. Treating decompilers as *software engineers*, and thus treating gotos as indicators of low-quality output, recent research into structuring algorithms has focused on extensively restructuring decompiled code to reduce or eliminate goto statements [9, 24, 44].

However, gotos *do* exist in popular C codebases, such as the Linux kernel, which, as of version 6.1, contains 3,754 gotos. Intuitively, previous work that removes all gotos will decrease the quality of decompiled code when it removes *intended, developer-written* gotos that are in the original source code, as

their elimination induces differences between the source code and the decompilation. Clearly, such gotos, *in the context of faithful decompilation*, are not harmful.

In fact, the lack of decompiler consideration for intended gotos is a symptom of a more general weakness in decompilation approaches. Existing techniques tackle the structure of binary code without a principled understanding of how this structure came to be. Our intuition is that an understanding of the differences between intended and spurious gotos in decompiled code will shed light on the binary constructs that hamper code structuring during decompilation.

This paper is the first work to investigate and identify the actual causes of spurious gotos. We systematically studied why spurious gotos appear in decompiled code, tracking them down to wide arrays of optimizations leveraged by the compiler. We investigated all compiler optimizations at level `O2` and below in GCC 9.5[1] on Coreutils and identified all optimizations that may introduce unstructurable code. To our surprise, some of these optimizations (responsible for around 17% of spurious gotos) are even enabled in *non-optimized* (`O0`) mode and cannot be easily disabled.

This investigation led us to a realization: To preserve the intended structure, including source-present gotos, and eliminate sources of gotos introduced by the compilation process, decompilers can leverage structuring algorithms informed by compiler-derived knowledge. Based on our findings, we created the first compiler-aware control flow structuring algorithm, SAILR[2], that uses compiler-derived knowledge to *deoptimize* binary code and make unstructurable code structurable. SAILR mirrors the compilation pipeline of GCC and precisely inverts goto-introducing transformations rather than using broad (and imprecise) goto-eliminating modifications leveraged by prior work.

SAILR generates decompilation that, compared to state-of-the-art techniques, contains significantly fewer spurious gotos while maintaining high structure similarity to the source code. For example, among other evaluations, we evaluated decompilers against 7,355 functions (containing 1,367 intended gotos) from 26 popular C Debian packages. The state-of-the-art Hex-Rays decompiler included in IDA Pro [3] emitted 6,115 gotos while achieving an average Graph Edit Distance (GED) of 22.52 between the decompilation and the source code. SAILR reduced the gotos count to 2,673 (we discuss the remaining gotos in Section 9) while maintaining an average GED of 22.64. DREAM, a state-of-the-art structuring technique that guarantees the elimination of *all* (including intended) gotos, emitted 0 gotos and significantly impacted the quality of the decompilation, achieving an average GED of 45.99. An optimal decompiler should have both the same number of gotos as the source and a GED score of zero.

Furthermore, while comparing SAILR against existing research, we realized that existing structuring techniques are either not open-sourced or have bit-rotted, hampering research in the field. To enable impartial comparative studies, we built a new decompiler for `angr`, which we refer to as the ANGR DECOMPILER hereinafter, and implemented SAILR as well as other previous structuring algorithms (namely, Phoenix [9], DREAM [44], and rev.ng's Combing [24]) in it. Due to its Python implementation, our decompiler can be significantly slower than other decompilers, however, we expect performance to improve as it is further developed. We hope the ANGR DECOMPILER will foster future decompilation research, improve the reproducibility of decompilation techniques, and make future work in this area easier to approach.

Contributions. This paper makes the following contributions:

- We identify the root causes of unstructurable code during the decompilation of C: compiler optimizations and knowledge gaps between the decompiler and the compiler.
- We design SAILR, a novel compiler-aware structuring algorithm that precisely inverts the root causes of goto-introducing compiler transformations to generate decompilation significantly closer to the original code.
- We build an open-source decompiler ANGR DECOMPILER, and then implement SAILR as well as three other structuring algorithms on our decompiler for an impartial evaluation.
- We design a new set of evaluation metrics to measure the quality of control flow structuring during decompilation by measuring how close it is to high-quality source code. Using our metrics, we evaluate SAILR and demonstrate its improved structuring performance over other structuring algorithms and state-of-the-art decompilers.

In the spirit of open science, we are *actively* developing the ANGR DECOMPILER and SAILR as open-source projects[3]. We have also open-sourced our evaluation framework, which includes the specific versions of `angr` and SAILR used in our evaluation[4].

2 Background & Motivation

Before diving into the technical details of SAILR, we introduce necessary background knowledge about compiling C programs (Section 2.1), decompiling C programs (Section 2.2), state-of-the-art control flow structuring algorithms (Section 2.3), and the motivation of our research (Section 2.4).

2.1 Modern C Compilation

Modern C compilation has two high-level phases relevant to the understanding of the decompilation process. First, the source code is "lifted" into an Intermediate Language (IL) as C is too high level to be operated on directly. In the case of GCC, C code is "lifted" into an IL called Gimple [2].

[1] We also analyzed LLVM and identified analogous optimizations.
[2] SAILR: Structured AIL Reverter.
[3] https://github.com/angr/angr
[4] https://github.com/mahaloz/sailr-eval

Once lifted, optimizations are applied to the IL. Optimizations that are specific to a program's control flow use *schemas* to identify optimization locations. Schemas are similar to puzzle pieces for a program's control flow. Once a schema is found to fit a certain control flow archetype, such as a while loop, the corresponding operation can be performed on the IL to manipulate it into an optimized form.

2.2 Modern Binary Decompilation

Modern binary decompilers generally perform three phases of analysis before generating C-style code.

Control flow graph recovery. A decompiler first disassembles machine code in a binary to recover assembly instructions, function boundaries, and execution flows. Control flow information is stored in a control flow graph (CFG), with basic blocks as vertices and execution flow as edges. Optionally, the decompiler may lift instructions into an IL to assist with architecture- and platform-agnostic analysis.

Type inference. After recovering a CFG, the decompiler uses parts of the graph to infer variable locations and their associated types. Prior research has explored various methodologies for inferring types, such as TIE [30], Howard [39], and retypd [34]. More recently, Osprey [46] and DIRTY [12] utilize machine learning techniques to predict types. Optionally, this inference phase may also recover prototypes and calling conventions for all functions.

Control flow structuring. In this last phase, the decompiler will transform assembly instructions or IL statements into C-style statements using patterns and flow information from the CFG. This phase will also simplify these statements and identify common compiler idioms to replace them with more human-friendly representations. In general, a CFG can be structured in many different, but semantically equivalent, forms, making choosing the right structure a difficult task.

2.3 Structuring in C Decompilation

Table 1 shows a taxonomy of existing control-flow structuring algorithms. We broadly categorize structuring algorithms into two groups:

Graph-schema-matching algorithms. These structuring algorithms attempt to match subgraphs of a CFG against known control-flow patterns for C control-flow structures. For example, a diamond-shaped subgraph commonly corresponds to an `if-else` construct when the two middle nodes have inverted edge conditions. Being condition-aware is critical to the correctness of these structuring algorithms [9].

A key challenge that these algorithms face is unknown graph schemas. Structuring algorithm authors usually hard-code graph schemas for common C control-flow constructs. However, because compiler optimizations create novel graph schemas, there are more graph schemas in real-world binaries than one can reasonably enumerate. As a result, these algorithms must virtualize edges (i.e., eliminating edges and replacing them with spurious gotos) to create more subgraphs that would match against known graph schemas, which significantly hampers the quality of decompiled code.

Goto-less algorithms. These structuring algorithms (used by DREAM and rev.ng) do not use edge virtualization to convert non-schema-matching subgraphs into schema-matching ones. Instead, they create new control-flow structures that are functionally equivalent to non-schema-matching subgraphs. These new structures often do not match the source's because they can introduce new or duplicated code. DREAM duplicates code found in conditions while moving around contained logic. rev.ng duplicates code found in incomplete structures to make them match known schemas. These algorithms rely heavily on block condition information to make goto-less structures.

Table 1: A taxonomy of control flow structuring algorithms in modern decompilers and decompilation literature. "Duplication Removal" refers to removing compiler-introduced code duplication. An optimal compiler will remove duplicates and have no redundant code/conditions. We observed that in rare cases Hex-Rays will remove exactly duplicate blocks.

Decompiler	Condition Aware	Emit Gotos	Duplication Removal	Open-source	Redundant Conditions	Redundant Code
Hex-Rays [3]	Yes	Yes	Rare	No	No	No
Ghidra [35]	Yes	Yes	No	Yes	No	No
mirtoc (extended) [21]	No	Yes	No	No	No	No
Phoenix [9]	Yes	Yes	No	No	No	No
DREAM [44]	Yes	No	No	Yes	Yes	No
rev.ng [24]	Yes	No	No	No	Yes	Yes
SAILR	Yes	Yes	Yes	Yes	No	No

```
1  int schedule_job(int needs_next, int fast_job, int mode)
2  {
3      if (needs_next && fast_job) {
4          complete_job();
5          if (mode == EARLY_EXIT)
6              goto cleanup;
7
8          next_job();
9      }
10
11     refresh_jobs();
12     if (fast_job)
13         fast_unlock();
14
15 cleanup:
16     complete_job();
17     log_workers();
18     return job_status(fast_job);
19 }
```

Listing 1: A motivating example based on code from the Linux kernel job scheduler.

```
1  long long schedule_job(unsigned int a0,
   ↪  unsigned int a1, unsigned int a2)
2  {
3      if (a0 && a1)
4      {
5          complete_job();
6          if (EARLY_EXIT != a2)
7          {
8              next_job();
9              refresh_jobs();
10         }
11     }
12
13     if (!a0 || !a1)
14         refresh_jobs();
15     if (a1 && (!a0 || EARLY_EXIT != a2))
16         fast_unlock();
17
18     complete_job();
19     log_workers();
20     return job_status(a1);
21 }
```

```
1  long long schedule_job(unsigned int a0,
   ↪  unsigned int a1, unsigned int a2)
2  {
3      if (a0 && a1)
4      {
5          complete_job();
6          if (EARLY_EXIT == a2)
7              goto LABEL_4012eb;
8          next_job();
9          refresh_jobs();
10         goto LABEL_4012d3;
11     }
12     refresh_jobs();
13     if (!a1)
14         goto LABEL_4012eb;
15 LABEL_4012d3:
16     fast_unlock();
17 LABEL_4012eb:
18     complete_job();
19     log_workers();
20     return job_status(a1);
21 }
```

```
1  long long schedule_job(unsigned int a0,
   ↪  unsigned int a1, unsigned int a2)
2  {
3      if (a0 && a1)
4      {
5          complete_job();
6          if (EARLY_EXIT == a2)
7              goto LABEL_4012eb;
8          next_job();
9      }
10     refresh_jobs();
11
12
13     if (a1)
14         fast_unlock();
15
16
17 LABEL_4012eb:
18     complete_job();
19     log_workers();
20     return job_status(a1);
21 }
```

Figure 1: (From left to right) the DREAM, Phoenix, and SAILR decompilation of Listing 1 (using GCC 9.5 -O2).

2.4 Motivation

At first glance, goto-less algorithms may appear superior to graph-schema-matching algorithms since they never generate spurious gotos. However, between the two structuring algorithms, goto-less approaches often differ more from the source's control flow structure. We assume that C source code in a popular and actively maintained codebase (such as code in GNU packages) is a high-quality implementation of program logic. Therefore, when decompiling binaries that are compiled from high-quality code, decompilation that is structurally close to the source is of high quality. Using source code structure as the ground truth, we manually evaluated modern structuring approaches and found their results to be significantly flawed.

Listing 1 shows a motivating example based on code found in the Linux kernel scheduler. Figure 1 shows the DREAM, Phoenix, and SAILR decompilation, respectively, for the compiled binary from Listing 1. While there are no spurious gotos in the DREAM decompilation, the `if` statements contain complex and irreducible conditions that make understanding execution flows difficult. DREAM decompilation contains duplicated conditions that did not exist in the source on Lines 13 and 15. Although this duplication decreases code complexity compared to Phoenix, it obfuscates the original programmer's intentions. It also makes the needed conditions at each line of code, like Line 16, harder to understand.

In the Phoenix decompilation, although it contains a goto found in the source code, it also contains two spurious gotos. The gotos targeting `LABEL_4012eb` are due to a lack of special-case `if`-statement schema in Phoenix. The goto targeting `LABEL_4012d3` is due to a lack of compiler-optimization knowledge. The compiler introduced extra code and created a path to skip an extra conditional instruction using the Jump Threading optimization (see Section 3.3). This optimization introduces an extra call to `refresh_jobs` in both the Phoenix and DREAM decompilation.

All the decompilers that we tested failed to remove the duplicated statements in this example. Additionally, any schema-based structuring algorithms, like those used in Ghidra and Hex-Rays, cannot remove the goto targeting `LABEL_4012d3`. This is because no schema exists to structure this goto edge. To generate high-quality decompiled code, we must develop a new structuring algorithm that can undo the compiler's transformations, remove the duplicated blocks, and generate high-quality decompilation that is close to the original source. The last code snippet in Figure 1 shows the decompilation result of such a new structuring algorithm.

3 Goto-Inducing Compiler Transformations

Without considering goto statements in source code, each C construct can be represented by a single-entry, single-exit (SESE) graph whose nodes are basic blocks in the construct [21]. Compilers rely on a finite number of graph schemas during the code generation phase (Section 2.1). If we split the binary CFG of a function into nested SESE graph regions, a graph-schema-matching structuring algorithm matches each graph region against a known schema as long as it knows *all* graph schemas that a compiler uses. This works for binaries built without *any* compiler optimizations.

Unfortunately, some compiler optimizations split, merge, or duplicate nodes on graphs without preserving graph schemas. These optimizations create new graph schemas and make it impossible for a structuring algorithm to exhaustively enumerate all possible graph schemas that an optimization compiler may generate. We find that there are three reasons for edge virtualization to happen during structuring:

- (Real) gotos that exist in the source code. An optimal structuring algorithm will want to preserve these gotos.

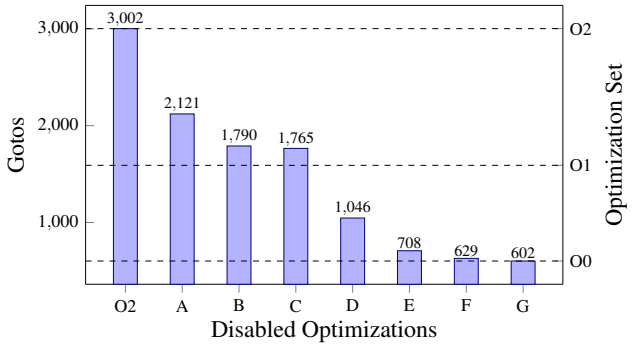

Figure 2: Gotos present in Hex-Rays decompilation as optimizations in Section 3.3 are disabled. Each optimization point disables itself and all optimizations to its left. Optimization sets `O2` through `O0` are shown for reference.

- Graph schemas that compilers use during code generation (without optimizations) but are not known to structuring algorithms. Because there are a finite number of these schemas, an optimal structuring algorithm should know all of them.
- New graph schemas that compiler optimizations create. An optimal structuring algorithm should remove all resulting gotos by reverting these optimizations.

We pick Coreutils 9.1 as the high-quality C codebase and GCC 9.5 as the compiler for our studies.

3.1 Enumerating Missing Graph Schemas

While one could read all of the source code of a compiler to find all unsupported graph schemas in a decompiler, this approach does not scale. We propose the following methodology to identify schemas missing in a decompiler.

First, we collect a set of binaries from high-quality codebases and compile them using GCC with optimizations (`O2`) while enabling the `save-temps` and `dump-tree-all` flags. These flags ask the compiler to save *intermediate files* that are generated during preprocessing and optimization passes. Next, we decompile all functions in all binaries and identify functions with gotos in the decompilation but not the source. For each function, we manually note the locations of the gotos in the function and perform a binary search on the intermediate files to identify which optimization pass changed the function to induce the goto. This identifies the GCC source for the optimization pass, and these passes are well-commented with information on the schema used.

3.2 Measuring Goto Introduction

Once we identify an optimization pass that causes an unstructurable subgraph, we attempt to understand the impact scale (i.e., how many gotos this optimization induces for binaries in Coreutils). We disable the optimization and repeat the steps in Section 3.1 until the goto disappears from the final decompilation. If the goto disappears, we count all the gotos in the decompilation before and after the optimization to measure its impact. We group all the disabled optimizations that are required to make the goto disappear as either one optimization algorithm or a set of tightly coupled algorithms.

This method worked surprisingly well for most optimizations in the GCC `O2` optimization set. It is worth mentioning that some optimizations did not disappear after disabling their frontend optimization option; These optimizations could only be entirely disabled by undocumented developer options.

3.3 Goto-Inducing Transformations in GCC

In an analysis of 1,150 unique functions across 135 object files in Coreutils 9.1, we found the cause of every goto across the 3,002 gotos that are present in the decompilation of Hex-Rays 8.0. We compiled these object files with GCC 9.5 with optimization level `O2`, inlining disabled (for function tracking), debug symbols enabled, and intermediate file dumping. We filtered these functions for uniqueness by eliminating functions that shared the same name, with the exception of common symbols such as `main` and `usage`.

We associate every goto to a specific GCC transformation using the methodology in Section 3.2. These transformations include togglable optimizations, internal highly-coupled optimizations, and compiler knowledge exploitation.

Jump Threading (A). When one branch's conditions superset a future branch's conditions, the statements contained between the two may be duplicated into the first and end with a jump to avoid extra conditional instructions. Gotos appear between duplicated statements.

Common Subexpression Elimination (B). The compiler finds common statements among multiple blocks and reduces them into one use of the expression and a series of jumps to that expression. Gotos occur between condensed statements.

Switch Conversion (C). The compiler replaces simple assignments on switch statements in cases with assignments from scalars. Gotos can occur between cases that share a common expression for assignment.

Cross Jumping (D). Unifies equivalent code (e.g., repeated statements) across regions and replaces duplicates with a jump to the unification. A goto corresponds to the new jump.

Software Thread Cache Reordering (E). The compiler estimates the likelihood of executing a set of paths and clusters those frequently executed together through code duplication.

Loop Header Optimizations (F). The compiler moves branches that are always true or true only once to avoid executing a conditional instruction many times. The edge leaving the loop body (for the copied header) becomes a goto.

Builtin Inlining (G). The compiler replaces special built-in functions, e.g., `strcmp`, with optimized inlining and propagation. Gotos can occur when inlining happens inside a short-circuit Boolean expression.

Switch Lowering (H). A highly-coupled optimization that is non-disablable in GCC. The compiler optimizes switch

statements by breaking them into clusters and applying heuristics to avoid large jump tables. Gotos occur when the switch statement is fully transformed into a nested `if` statement.

Nonreturning Functions (I). Some functions, such as `exit` and `abort`, may not (always) return, and GCC uses this knowledge to transform the CFG. When the decompiler lacks knowledge of these functions' ability to not return, it can cause successors to be incorrectly added during CFG recovery, causing goto edges to those successors.

Figure 2 shows the number of gotos present in decompilation after disabling transformations in compounding order. In addition to associating the cause of every goto, we group the goto-inducing optimizations into two classes by effect. Both classes broadly transform *irreducible statements*, which we define as statements found in the source code that cannot be eliminated. Such statements must always exist for the *same path* they were found in the source, though the literal instructions that cause them may be condensed.

Irreducible Statement Duplication (ISD) converts a statement into many statements that are semantically equivalent to the original. This set includes optimizations A, E, and F.

Irreducible Statement Condensing (ISC) converts many statements into a condensed version with introduced graph edges. This set includes optimizations B, C, and D.

Optimizations G, H, and I do not fit well into either ISD or ISC optimizations and instead are classified as miscellaneous optimizations. These miscellaneous optimizations account for a minority of goto-inducing optimizations and require more specific algorithms to revert.

4 Overview of SAILR

We implemented our structuring algorithm, SAILR, and our decompiler, ANGR DECOMPILER, on top of the `angr` binary analysis platform [38]. As such, our decompiler supports any architecture `angr` supports. The ANGR DECOMPILER handles miscellaneous decompiler tasks such as program lifting while SAILR structures lifted Control Flow Graphs (CFG).

4.1 The Decompilation Pipeline

The ANGR DECOMPILER is similar in design to prior work [9, 44]. Using `angr`, we convert a binary into a lifted CFG. The lifted CFG contains VEX Intermediate Representation (IR) instructions, which is the IR `angr` uses. Figure 3 illustrates the high-level steps in the ANGR DECOMPILER after a VEX IR CFG is provided.

4.1.1 Lifting, Simplification, and Variable Recovery

The ANGR DECOMPILER takes a function-level VEX IR CFG as input. Because VEX IR cannot represent C-style expressions, we designed a more abstractable intermediate language called the ANGR INTERMEDIATE LANGUAGE (AIL). The ANGR DECOMPILER starts by lifting the VEX IR CFG into an AIL CFG. Multiple rounds of simplifications are run on the AIL CFG to eliminate redundancy. This lifting process is similar to GCC's simplification process while lifting C to Gimple. After lifting, the ANGR DECOMPILER recovers variable locations, function prototypes, and variable types using the AIL CFG.

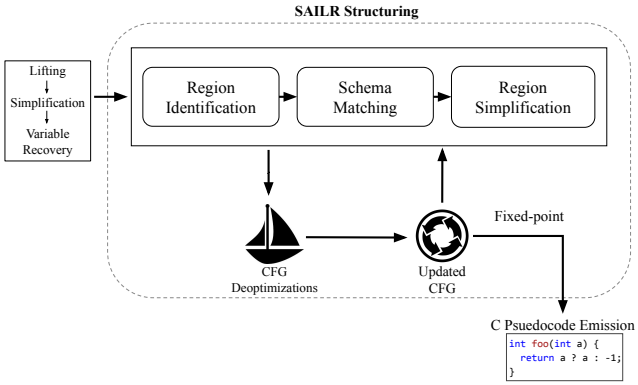

Figure 3: Overview of ANGR DECOMPILER's decompilation pipeline.

4.1.2 SAILR Control Flow Structuring

After collecting an AIL CFG and variable information, the ANGR DECOMPILER uses the SAILR structuring algorithm to turn the AIL CFG into C pseudocode. The following four passes are run until a fixed point is reached. These passes together make up the core of the SAILR structuring algorithm.

Region identification. The region identification algorithm in the ANGR DECOMPILER is based on the algorithm that DREAM employs [44]. First, it identifies all single-entry, single-exit (SESE) regions in an AIL graph, following a reverse topological order of all nodes. As its name suggests, a SESE region refers to a group of AIL blocks and edges that contain one entry and one exit. A region may contain multiple regions.

Schema matching. The ANGR DECOMPILER then preliminarily structures each region using a graph-schema-matching algorithm built using concepts developed in Phoenix [9]. A major difference between SAILR and Phoenix is that Phoenix directly matches schemas against subgraphs on the full graph while SAILR matches schemas on a per-region basis. SAILR matches subgraphs in each region against known graph schemas.

When SAILR fails to match a region against any known graph schemas, it will virtualize an edge by removing it from the region and replacing the source jump statement with a goto. Then, SAILR condenses blocks that match a known graph schema into a C-style construct. Once the ANGR DECOMPILER finishes structuring a region, it replaces the region in its parent region with the structured node before continuing

to structure its parent region. Structuring terminates once the outermost region is structured.

Region simplification. Next, the ANGR DECOMPILER performs simplification and optimization on the AIL graph of each refined region to eliminate redundant edges and blocks that may not affect the control flow. This is also applied to the preliminarily schema-matched result.

Deoptimization. In the last step of SAILR structuring, the ANGR DECOMPILER performs deoptimizations directly on the AIL graph. The deoptimization phase relies on all knowledge previously collected in the pipeline, such as region information, goto locations, and reaching conditions. Each deoptimization described in Section 5 is run iteratively or until a limit is met.

4.1.3 C Pseudocode Emission

Finally, the ANGR DECOMPILER expands the structured regions into a C-style abstract syntax tree (AST) and pretty-prints it as a function with linear C-style statements. One novel aspect in the decompilation pipeline is that we stall optimizations on the lifted function CFG until the last point of the process. This allows CFG edits, which are the basis for SAILR, to have the most available knowledge.

4.2 Major Contributions of SAILR

Previous works in control flow structuring have focused on making decompilation more structured [9, 24, 44]. The Phoenix [9] decompiler identified this unstructurability through gotos. Phoenix focused on reducing these gotos through condition-aware graph schema matching. DREAM [44] eliminated gotos by duplicating conditions across the code to create bounded statements. rev.ng [24] also eliminated gotos through duplication, but instead duplicated executable code, not conditions on nodes. In the case of DREAM and rev.ng, these approaches generically eliminate all gotos regardless of their existence in the source.

In contrast, SAILR does not blindly eliminate gotos and instead treats the cause: compiler transformations. SAILR precisely reverts compiler transformations found to be the cause of unstructurable code, which manifest as gotos in decompilation. Of these transformations, certain compiler optimizations and the gap between the decompiler and the compiler play a significant role in unstructurability. SAILR approaches a solution to both of these problems by improving the knowledge of the decompiler and reverting certain optimizations.

Reverting compiler optimizations. SAILR's deoptimization passes revert the most impactful optimizations to control flow structure. At a high level, SAILR directly edits the AIL graph by condensing (Section 5.1), duplicating (Section 5.2), or moving (Section 5.3) nodes. In each case, SAILR uses information derived from GCC 9.5 to perform the reverse actions of these optimizations. For each deoptimization pass described in Section 5, the majority of computation time comes from searching for valid cases. This search phase is *goto-guided*. We assume all optimizations that SAILR wants to deoptimize are goto-inducing, so we only search the blocks around a goto edge when finding candidates. Although the understanding of these optimization techniques is based on GCC, we find that other compilers, such as Clang, implement the same optimizations in similar ways.

Closing the compiler-to-decompiler knowledge gap. We identified two critical knowledge gaps by studying the Hex-Rays decompilation of Coreutils `O2` binaries compiled by GCC 9.5. The first gap is the *returning* of a callee, i.e., whether a callee function returns or not. A compiler will not generate any return site for a call if it knows the callee never returns, for example, *_exit()*. The function CFG that any decompiler recovers will be different from the compiler's CFG if they are not aware of this fact, which may lead to unstructurable code. This problem is further complicated because the returning can be call-site specific: A callee function may be returning in one call site and non-returning in a different site, which we will detail in Section 6.1. The second gap is the decompiler's unawareness of all the graph schema the compiler uses. We discuss major ones in Section 6.2.

5 Deoptimizing Decompilation

In this section, we give an overview of novel algorithms to deoptimize decompilation. These algorithms revert three types of optimizations discussed in Section 3.3: ISD, ISC, and miscellaneous optimizations. In each deoptimization, there is a *search* and *transformation* phase that parallel the compiler's optimization process. The search phase is generally more computationally intensive.

5.1 ISD Optimizations

The main effect of ISD optimizations is the duplication of subgraphs found in the original source CFG. Most compilers will limit the number of statements that can be duplicated when performing optimizations in this class. During GCC's ISD optimization passes, reaching conditions are computed for each statement in the CFG. Reaching conditions determine what statements are accessible when a condition holds during execution. For each optimization, GCC further determines if a statement is duplicatable using other indicators. In the case of Jump Threading, the indicator is whether an ancestor statement shares an overlapping condition that guarantees the execution of the current statement. For optimizations such as Software Thread Cache Reordering, the duplication indicator can be as complicated as computing if a block has a higher execution probability over its neighbors.

Next, the compiler will duplicate both the marked statement and the graph found under that reaching condition. The duplication will be placed at another location in the graph that guarantees the same reaching conditions, guarded by either

```
void foo(int a, int b) {
  if (a && b) {
    puts("first print");
  }

  puts("second print");
  if (b) {
    puts("third print");
  }

  sleep(1);
  puts("leaving foo...");
}
```

```
void foo(int a, int b) {
  if (a && b) {
    puts("first print");
    puts("second print");
    goto label_1;
  }
  puts("second print");
  if (b) {
label_1:
    puts("third print");
  }
  sleep(1);
  puts("leaving foo...");
}
```

Figure 4: Example C code shown before and after transformation from Jump Threading, an ISD optimization. The second condition of the original code is always true if the first condition is true, causing the comparison to be subverted by a jump.

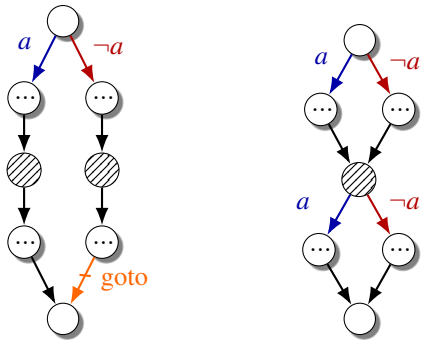

Figure 5: CFGs before and after deoptimizing an ISD optimization case. In order to identify an ISD case, the shaded node must be found to have a semantic duplicate, as well as post-dominating goto edge. The nodes are merged and then bounded by their previous conditions.

already present blocks or new conditional blocks. The duplicated subgraph may have other statements appended to the graph tail nodes. Finally, the duplicated subgraph tail nodes are connected to the original subgraph successors by an edge.

Figure 4 shows a C program that when compiled with GCC `O2` optimizations results in optimized output. The compiler computes the conditions for both the first `print` and the third `print`. Because the conditions overlap, the compiler determines that both scopes will execute if the first condition (*a* && *b*) holds true. The compiler then duplicates the code between the two conditions and creates a jump (a goto) from the first condition scope to the second condition scope. We next discuss how SAILR condenses ISD-duplicated blocks.

Finding duplicate statements. Given an AIL function CFG, SAILR starts by iterating through all combinations of two statements and creates an initial set of candidate statement pairs that exist in the same region and are *storage equivalent*. Two statements are storage equivalent when all their reads and writes to locations are the same. For a pair of call statements, storage equivalence means all arguments share the same storage locations and the function addresses are the same.

With the candidate set, we take all candidates and expand their similarity match by expanding the connected graph of the matching statements. We assume the first found statement of the pair is the head of each similar subgraph. Then we iteratively expand each subgraph using a Breadth-First Search (BFS) traversal and match the statements using the Knuth-Morris-Pratt (KMP) algorithm [28].

Finally, with each candidate expanded to its maximum graph similarity, we find the tail nodes of each duplicate graph. All of these nodes must share the same successor in each graph. At least one of the two graphs must have a goto edge, which was identified during structuring, that connects it to the common successor. In Figure 5 (left) the graph of a matching case is shown. Note, that any number of nodes can exist between the condition head and the duplicated node.

Condensing duplicates. The key to condensing a pair of candidate subgraphs while preserving the original semantics is maintaining the reaching conditions of every node in the graph after condensing. In Figure 5 (right), a graph shows the correct result of reverting an ISD case.

First, the duplicated node must be merged into one node by redirecting all predecessors to the newly merged node. Although shown as a single node in this example, the merged node can be a graph, which means the predecessors may go to multiple nodes. Next, the condition *a* must be used to correct the flow of execution before and after leaving the merged node. If there are multiple head nodes in the merged sub-graph, each head node must have the correct *a* condition blocking its path. In the case of a single node being the head of a merged graph, only the exits of the merged sub-graph need duplicated conditions. Finally, the graph is in the deoptimized state and should no longer have a goto. Depending on the conditions before and after the ISD case, we can simplify the exiting conditions on the merged sub-graph. Listing 3 in the Appendix shows the pseudocode of our ISD deoptimization algorithm.

5.2 ISC Optimizations

The main effect of ISC optimizations is the reduction of subgraphs found in the original source CFG through condensing. Condensing turns n equivalent statements into $n-m$ statements, where $m \geq 1$. Similar to ISD optimizations, ISC optimizations require knowledge of reaching conditions and statement semantics. ISC optimizations traverse the CFG for semantically equivalent pairs of statements (or subgraphs). Then, for each statement pair, ISC optimizations will eliminate one statement and connect the other with a jump to the remaining statement.

Figure 7 shows an example C program that exhibits gotos when optimized with Cross Jumping, an ISC optimization. The `return` statement is reused across many if-statements,

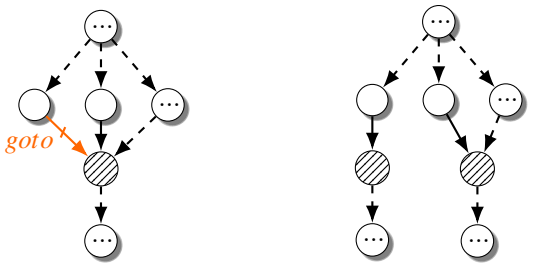

Figure 6: CFGs before and after deoptimizing an ISC optimization case. ISC cases contain a goto edge connecting one node to another node that has multiple predecessors. Duplication of the shaded node, and all its single-successor ancestors, revert this case.

```
int foo(int a, int b) {
  if(!a)
    return -1;
  puts("first print");
  if(!b) {
    return -1;
  }

  puts("leaving foo...");

  return 1;
}
```

```
int foo(int a, int b) {
  if(!a)
    goto label_1;
  puts("first print");
  if(!b) {
label_1:
    ret = -1;
    goto label_2;
  }
  puts("leaving foo...");
  ret = 1;
label_2:
  return ret;
}
```

Figure 7: Example C code shown before and after transformation from Cross Jumping, an ISC optimization. In the original code, the *return* statement, as well as its return value, are reused in the same execution pass resulting in statements being merged and connected with a goto.

which is a common programming pattern. In the resulting code output, all but one `return` is replaced with a goto. For the first `if`, Cross Jumping also eliminates the constant assignment to `ret`. We next discuss how SAILR duplicates ISC-condensed blocks to recover the original structure.

Finding condensed statements. Finding condensed statements requires finding the locations of goto edges (after structuring) in an AIL CFG. Similar to searching for ISD cases, this demands that a decompiler can iteratively perform graph optimizations and structuring. We can only know the locations of gotos in an AIL function CFG after structuring at least once. Figure 6 (left) shows a matched case of an ISC optimization.

Our algorithm considers any statement connected to another statement through a goto as an ISC deopimization candidate. ISC deoptimizations are a wide-catching net for gotos and may not always eliminate gotos. To avoid re-duplicating blocks that other passes condensed, this deoptimization must be run after all other goto-aware deoptimizations. As a special case, gotos to function-exit blocks, such as a return, are special-cased to reduce duplication. An arbitrary code duplication limit is set on jumps to these types of blocks to mimic the compiler. This allows the `goto cleanup` programming style to still exist in SAILR decompilation.

Reverting condensed statements. After locating all goto edges in the AIL graph, the destination nodes act as the head nodes for duplication. To revert an ISC optimization, the chain of single-successor nodes is copied beginning from the head node. Figure 6 (right) shows the effect of the duplication on an ISC case. For a node chain that ends in an exit node, or a node that leaves the current function, it is guaranteed to remove a goto. Listing 4 in the Appendix shows the pseudocode of our ISD deoptimization algorithm.

A common case we observe is that many functions have two exit nodes after a comparison of the stack canary. Therefore, we identify all the exit regions, or clusters of nodes, that end in an exit that has one entry. For stack canary comparison, there are three nodes in the cluster: the if statement, the return node, and the stack-check-fail node. This allows us to copy the entire region as the exit node.

In cases where the final node in the node chain is not an exit node, this deoptimization may not remove the goto. Instead, this deoptimization transforms the graph into a form that is easier to case match for other deoptimizations that run on the graph. This is important because during compilation, ISC optimizations are often stacked on top of other goto-inducing optimizations, similar to the ISD optimizations. SAILR must iteratively revert these optimizations one after another.

5.3 Miscellaneous Optimizations

Optimizations not deoptimized by ISD or ISC reverters require specific identification. Of the known miscellaneous optimization, only Switch Lowering may be identified using gotos.

Reverting switch lowering. To the best of our knowledge, depending on the density of switch case numbers, GCC may generate three forms of binary control-flow structures for switches: binary decision trees, jump tables, and bit-tests [4]. When fallthroughs between cases are in use, switches that are lowered into decision trees are usually unstructurable as cascading `if-else` constructs. To make such subgraphs structurable, we must transform them into normal switches. In SAILR, we implement a solution based on pattern and condition matching to revert lowered switches. Among all decompilers that we tested, only Hex-Rays has very limited support for reverting lowered switches. In Figure 8 a case of cascading `if-else` constructs is shown as well as its transformation into a switch node. A goto present in the cluster is optional but is a high-confidence indicator of a lowered switch.

Reverting switch clustering. Another common cause of switch-related unstructurable subgraphs is switch clustering [4], where a switch is broken into multiple disjoint subswitches (with different case numbers). Each sub-switch is transformed into its own binary representation (of the three

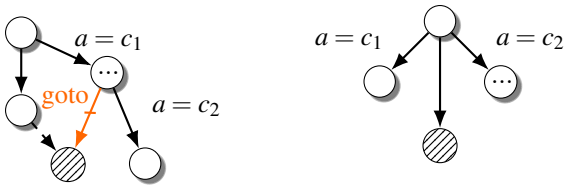

Figure 8: CFGs before and after Switch Lowering reversion. All nodes in a candidate switch region must hold a Boolean condition of $a = c$, where c is a constant. These condition nodes must cascade and either be post-dominated by a single node or exit early. A goto appearing in the region makes the case more likely to be a real Switch Lowering case.

forms). SAILR identifies and merges these sub-switches that are grouped together by conditions within the same region.

6 Narrowing Knowledge Gaps

When the decompiler and the compiler diverge in their views of the same binary, it may cause otherwise structurable code to be incorrectly regarded as unstructurable by the decompiler. Therefore, we must close or narrow the knowledge gaps between the decompiler and the original compiler to ensure they have the same view of all structuring-dependent artifacts. Our intuition is that decompilers must be *compiler-aware* to achieve high-quality decompilation. In this section, we summarize two significant sources of knowledge disparity that we identified (described in Section 3.3) on GCC-9.5-compiled Coreutils binaries: The returning of functions and missing graph schemas in decompilers.

6.1 Non-returning Functions

GCC does not generate edges or successors for function calls (e.g., `exit()`) that do not return. A decompiler that is not aware of these semantics may incorrectly add a spurious edge from the return site (that does not actually exist) to whatever block that follows the call. Unfortunately, fixing this problem requires more than just providing decompilers with a list of non-returning functions, because a function may be returning in some call sites and non-returning in other call sites.

The `error` function from glibc is an example: It returns when the first argument (`status`) is 0, and does not return in other cases. With the `error.h` header file, GCC determines that any call sites for `error` do not return if the first argument has a non-0 constant. To solve these, we augmented the ANGR DECOMPILER so that it determines the returning of `error` call by examining the value of its first argument at all its call sites.

6.2 Additional Graph Schemas

Schema-matching algorithms, as described in Section 2.3 match subgraphs against known graph schemas that correspond to C structures. By using the methodology outlined in Section 3.1, we identified three major missing graph schemas in most decompilers that we tested and added support for them in SAILR.

Short-circuited compound Boolean conditions. During the lifting of C to Gimple, GCC transforms compound Boolean conditions into control-flow constructs using specialized graph schemas to express the short-circuit condition. Both Hex-Rays and Ghidra support such graph schemas, but Phoenix does not and always generates gotos. Moreover, DREAM will generate a semantically equivalent graph where each node is guarded by much more complex conditions than what are in the source. We investigated and concluded that the root cause is DREAM's reliance on condition expression simplification. Simplification (or minimization) of Boolean expressions is an NP-hard problem, so it is too computationally expensive for DREAM to derive Boolean expressions that are as simple as in the source for non-trivial cases (e.g., Boolean expressions with more than six atoms).

Comma-separated multi-statement expressions. A not-that-uncommon usage in C is comma-separated multi-statement expressions, where the statements will be executed first before evaluating the expression at the end. Not supporting this expression type will prevent proper structuring of loops that use such expressions as loop conditions. Out of all decompilers that we test, only Hex-Rays supports such types of expressions. We add support for these expressions in SAILR.

Common condition while-loop exits. In some cases of structuring, the Phoenix algorithm can generate extra gotos that leave cyclic regions structured as `while-loops`. In a `while-loop`, if the loop contains a branch where both successors are inverted conditions of each other, like c and $!c$, this can be replaced by an `if` statement and a break out of the loop.

7 Measuring the Quality of Structuring

Measuring the structuring quality of decompilation is different from measuring its overall quality. Changes that improve readability, e.g., constant propagation or variable elimination, do not always impact the structure of the code. Previous work focused on evaluating the quality of structuring by relating it to structural complexity. They mainly used three metrics: Number of gotos (Gotos) [9, 24, 44], McCabe Cyclomatic Code Complexity (MCC) [24], and the Lines of Code (LoC) [44]. However, these metrics are flawed for evaluating structuring quality even when shown relatively similar source.

To understand these flaws, we evaluate previous metrics on our motivating example in Listing 1, with the decompilation results by Phoenix, DREAM, and SAILR shown in Figure 1

Table 2: Previous work's structuring metrics, GED, and CFGED measured on Figure 1 and Listing 1.

	Gotos	LoC	MCC	GED	CFGED
Source	1	19	4	0	0
SAILR	1	15	4	0	0
Phoenix	3	19	4	2	2
DREAM	0	16	9	21	38

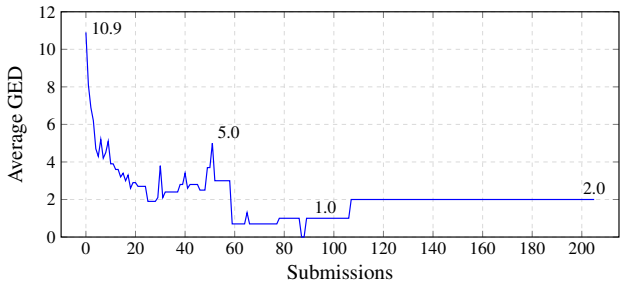

Figure 9: In the Decomperson study, a human participant's goal is to perfectly recover the source from only the assembly code of a binary. As human participants submit closer byte-matched decompilation, their average GED scores decrease.

(and measurements produced in Table 2). SAILR achieves perfectly structured decompilation, but its MCC is identical to Phoenix's result, despite the latter producing differently structured code from the source. This is because MCC only measures the number of independent paths without considering the resemblance of code structures. As a result, MCC can return significantly lower scores for decompiled code with spurious gotos rather than properly structured conditions because conditions add compounding edges in the CFG. For gotos, although Phoenix's decompilation was structurally closer to the source than DREAM's decompilation, Phoenix's score is worse than DREAM's score. Lastly, LoC rewards decompilers for non-structurally related optimizations, such as aggressive expression folding and constant propagation.

7.1 Graph Edit Distance

Conceptually, MCC is flawed because it does not consider edge and node locations relative to the source. Using this flaw as inspiration, we used the Graph Edit Distance (GED) relative to the source CFG to encode edge-node location differences into a metric. Prior work has used GED in binary similarity analysis by computing GED on binary-level CFGs [7, 42]. We run GED on the CFGs of the source and associated decompilation to estimate their structural closeness.

We associate lower GED-to-source scores with easier-to-understand decompilation. To study this association, we collected results from Decomperson, a study of human-written decompilation [11]. In Decomperson, participants manually create decompilation given assembly with the target of having the lowest assembly difference from the binary when compiled. With every submission, users attempt to get closer to the source code.

In Figure 9, we graph the average exact GED score of 30 participants as they submit results on the `winky` challenge. We could compute the exact GED scores because the functions in this challenge were sufficiently small. Each participant ended with a byte-match of 50% or higher. Their average GED scores decrease over time, indicating lower GED scores are likely associated with higher closeness to source, which means fewer structural differences.

7.2 Control-Flow Graph Edit Distance

Unfortunately, computing an exact GED is NP-hard [10]. In practice, we have found exact GED is usually too expensive to run on graphs with more than 12 nodes. When using upper-bound GED estimation algorithms, their estimations are significantly higher than the exact score, and, on exact match graphs, they rarely return zero. Therefore, we design a scalable algorithm, called Control-Flow Graph Edit Distance (CFGED), that uses CFG-specific characteristics to improve the scalability of GED estimations when applied to decompilation.

CFGED decomposes the GED problem using the identified SESE regions and uses exact GED when the region is small enough, and approximate GED if the region is too large (as the last resort). The CFGED between two CFGs is the sum of the GED of all regions. This often returns a score that is lower than the upper-bound GED approximation in the same or faster time. An example of CFGED computation on two graphs can be found in Appendix A.3.

To validate the accuracy of CFGED, we computed the exact GED, the maximum GED, and the CFGED of all functions with lower than 12 nodes in Coreutils 9.1. On all 506 samples, CFGED returns a score between the maximum and exact. CFGED gets the exact score on 64% of all samples. On average, CFGED introduces a 35% overhead to the percent difference score (the GED approximation divided by the max possible score).

8 Evaluation

We evaluate all structuring algorithms using two sets of C binaries: popular Debian packages and MSVC compilable packages. We attempt to answer the following research questions through experiments:

RQ1. How well does SAILR structure function CFGs during binary decompilation compared to state-of-the-art structuring algorithms?
RQ2. Given that SAILR is guided by findings on GCC 9.5, how does it perform on older and newer GCC versions?
RQ3. Given that SAILR is guided by findings on GCC 9.5, how does it perform on binaries generated by different C compilers?

Table 3: Structuring results on 7,355 functions across 26 popular Debian packages. The percent change relative to source is shown on each sum. The CFGED percent change is shown w.r.t. Hex-Rays.

Metric	Source			SAILR			Hex-Rays			Ghidra			Phoenix			DREAM			rev.ng		
	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med
Gotos	1,367	0.19	0.0	2,673 (95.5%)	0.36	0.0	6,115 (347.3%)	0.83	0.0	6,575 (380.9%)	0.89	0.0	8,497 (521.6%)	1.16	0	0 (100%)	0	0	0 (100%)	0	0
Bools	6,180	0.84	0.0	3,980 (35.6%)	0.54	0.0	4,279 (30.8%)	0.58	0.0	4,850 (21.5%)	0.66	0.0	2,685 (56.6%)	0.37	0.0	43,661 (600.5%)	5.94	0.0	2,003 (67.6%)	0.27	0.0
Calls	53,995	7.34	3.0	52,558 (2.6%)	7.15	3.0	52,508 (2.8%)	7.14	3.0	53,202 (1.5%)	7.23	3.0	51,167 (5.2%)	6.96	3.0	51,204 (5.2%)	6.96	3.0	166,798 (116.3%)	22.68	3.0
CFGED	0	0	0	166,468 (0.5%)	22.64	8.0	165,583 (0%)	22.52	8.0	187,509 (13.2%)	25.5	7.0	166,480 (0.5%)	22.64	8.0	338,231 (104.3%)	45.99	10.0	524,248 (216.6%)	71.29	8.0

Additionally, we explore tangentially related questions with an extended evaluation set in Appendix A.4.

8.1 Evaluation Methodology

All experiments are performed on an Ubuntu 22.04 server. During each experiment, we compile all .c files in a package and use the compiler to output intermediate source files that have been preprocessed. Next, we compile with debug symbols, no inlining, and saved object files. We then record the address-to-line mapping for each object file before stripping it of debug information. The mapping is recorded for evaluation with CFGED. Each object file has an associated intermediate (.i) file and address mapping.

After compiling, we decompile every object file, which is the non-linked version of the final executable, with every decompiler and parse the output with Joern [45]. We use Joern to collect metrics, as well as CFGs, from each decompiler to allow for decompilation that may not be recompilable. We evaluate Hex-Rays 8.0 (IDA Pro), Ghidra 10.1, and the ANGR DECOMPILER, which implements SAILR, Phoenix, DREAM, and rev.ng's Combing. Each structuring algorithm is evaluated on the metrics in Section 8.2. To assure a fair evaluation among all structuring algorithms, we only report metrics on functions that successfully decompiled on all decompilers and did not crash on our measuring pipeline.

Reimplementation of existing algorithms. For fair evaluation, we attempted to use prior work's decompilers. However, the Phoenix decompiler was not open-sourced, and the open-sourced DREAM decompiler does not build in their specified environment. We tried to use the rev.ng decompiler, but the alpha version crashed on most binaries in the Coreutils package, which prohibited us from evaluating it. We reimplemented all of them in the ANGR DECOMPILER.

8.2 Metrics

We evaluate the structuring algorithm of every decompiler using two metrics.

Structuredness. Structuredness measures how often a structuring algorithm creates C structures in the output. We follow prior work evaluations [44]: we measure structuredness by counting the number of gotos in the decompilation.

Faithfulness. Faithfulness measures how close the control flow structure is to the original source's. We use CFGED, function calls, and Booleans expressions to measure the similarity in flow, execution structure, and conditions to the source.

We further ensure the structuring correctness of each decompiler's output by manually sampling and verifying a small set of decompiler output. We do not measure the correctness by recompilability because the main contribution of our work is structuring, not recompilable decompilation.

Table 4: Structuring results on 433 functions across Coreutils compiled with various GCC versions and Clang.

	Most-Recent Release	Decompiler	Gotos	Bools	Calls	CFGED
Source	N/A		20	438	4,761	0
GCC 5	October 10, 2017	SAILR	152	295	4260	14499
		Hex-Rays	464	299	4199	14399
GCC 9	May 27, 2022	SAILR	169	284	4277	13462
		Hex-Rays	447	290	4353	13266
GCC 11	April 21, 2022	SAILR	170	280	4290	13445
		Hex-Rays	451	293	4353	13391
Clang 14	March 25, 2022	SAILR	167	292	4290	22879
		Hex-Rays	454	303	4335	22671

Table 5: Structuring metrics on 45 unique functions across Zlib compiled with MSVC.

	Gotos	Bools	Calls	CFGED
Source	0	70	158	0
SAILR	12	59	133	1813
Hex-Rays	14	52	133	1743

8.3 Structuring on Debian Packages

To answer RQ1, we focused on the effects of SAILR concerning programs compiled with GCC 9.5 on the `O2` optimization level, the compiler configuration we studied. We evaluate against all structuring algorithms on 26 popular Debian C packages [1], listed in full in Appendix A.1. Table 3 shows the results on each metric for every decompiler.

SAILR outperforms all algorithms in relative goto creation but loses to Hex-Rays in CFGED by 0.5%. Although SAILR loses in CFGED, the high reduction of gotos relative to Hex-Rays indicates SAILR is removing gotos in a non-structurally destructive way. In contrast, DREAM, which removes the most gotos, differs from Hex-Rays by 104.3% on CFGED.

8.4 Generalizability Across Compilers

To answer RQ2 and RQ3, we evaluated SAILR on various compiler versions and vendors. Building on the results of Table 3, we focus on the differences between the best structuring

algorithms of the Debian package evaluation: Hex-Rays and SAILR. In Table 4 we evaluate multiple versions of GCC as well as Clang on Coreutils, the defacto decompilation evaluation dataset [9, 24, 44]. Although Coreutils has a Windows variant, it does not natively compile with MSVC. In lieu of Coreutils, we chose Zlib since it was easily compilable. In Table 5 we evaluate Hex-Rays and SAILR on Zlib, compiled with MSVC 14.20.

Similar to the results in Table 3, SAILR marginally loses to Hex-Rays on CFGED (no more than 1.5%), but improves the gotos emitted across every compiler. Because the number of gotos, calls, and Boolean operators are all fairly consistent, we can conclude that the compiler-aware decompilation improvements that SAILR has on GCC 9 actually generalize to historical versions of GCC. However, in the highly different MSVC compiler, SAILR loses significantly more on CFGED, which may be due to hardcoded Windows-specific schema in Hex-Rays.

9 Discussion

To the best of our knowledge, this is the first effort for a complete investigation of goto-inducing compiler transformations and the significance of undoing such transformations in decompilers. We next discuss the limitations of SAILR.

The remaining gotos. While SAILR removes a lot of spurious gotos in decompilation, it does not remove all of them. Through a manual analysis of the remaining gotos, we find that they are caused by the following reasons: (a) Certain function calls not returning but angr is unaware, which leads to incorrect function-level CFGs; (b) The decompiler choosing the wrong edge among several candidates to virtualize, which prevents SAILR from applying deoptimization techniques at intended locations; and (c) SAILR condensing blocks that were not impacted by ISD optimizations in compilers. We believe (b) and (c) can be addressed by SAILR performing an iterative search when picking edges to virtualize and parts of code to deoptimize, which we leave as future work.

High CFGED between the best decompilation and source. We also investigated why the CFGED scores are still so high, even in cases where SAILR perfectly reverts every goto-inducing optimization that the compiler introduces. There are two main reasons. First, CFGED is still an approximation, which can significantly differ from the actual exact GED on larger CFGs. In a CFG with over 200 nodes and 100 edges, we observed that the exact GED between the decompilation and the source was as low as 4 while CFGED reported 306. We envision that future algorithmic improvements on GED computation will reduce GED scores in our measurement. Second, CFGED is extremely sensitive to structural differences, and many structuring choices that a decompiler makes (e.g., whether a node belongs to the loop body or not), while leading to high structural differences, are ultimately undecidable. We believe the use of statistical techniques, e.g., machine learning, will help immensely in these cases.

Architecture, platform, and compiler-specific study. Our investigation is specific to x86-64 Linux user-space executables that are compiled using GCC or Clang, and some MSVC-compiled Windows binaries. While it is likely to find new types of goto-inducing compiler transformations under other settings, decompiler authors can use the methodology we presented in Section 3.2 to find and address them.

Compounding ISC and ISD optimizations do not always cause gotos. In rare cases, compounding ISC and ISD optimizations may create subgraphs that happen to match against known graph schemas. As such, no unstructurable code regions will be created. A graph-schema-matching decompiler will generate a goto-free decompilation result, however it does not structurally match the source. Fundamentally, this is caused by the many-to-one mapping from legitimate C source to binary code. We believe a natural solution is using a machine-learning approach to predict the most suitable structuring result from the binary CFG and its context.

Using Hex-Rays to study gotos in Coreutils binaries. In the goto-origin study, we used Hex-Rays to investigate gotos in binary code with the assumption that Hex-Rays only performs basic graph-schema-matching and does not proactively remove gotos. We found that this assumption is false because Hex-Rays seems to have more graph schemas than other decompilers. Fortunately, we did not notice any cases where Hex-Rays would actively solve ISC- or ISD-optimizations. We believe using a naive graph-schema-matching decompiler can help increase accuracy of all goto sources.

10 Related Work

Control-flow Structuring. The two main methods for recovering control-flow structures from CFGs are *interval analysis* and *structural analysis*. Interval analysis [5, 16] partitions a CFG into nested regions called intervals. The nested nature ensures the data-flow analysis is not duplicating effort.

Structural analysis [37] is an extension of interval analysis, allowing a syntax-driven method of data-flow analysis that is normally reserved for abstract syntax trees to be applied on intermediate representations of low-level code such as AIL, VEX, and BIL. Structural analysis algorithms recover high-level control constructs from CFGs for use in decompilation.

Engel et al. [21] extended structural analysis to C-specific control statements, by proposing the SESS model. SESS accounts for statements in C that appear before control-flow changes induced by a `break` or `continue`. However, this approach relies on pre-defined schemas that occur in CFGs and introduces gotos that were not originally present in the source code. The SAILR approach focuses specifically on GCC-induced *gotos*.

A related area of research attempts to remove gotos at the source code level [22, 41] by defining AST transformations

that replace gotos with an equivalent semantic-preserving structure. This generally increases overall code size and complexity and also can insert unnecessary Boolean variables.

Decompilers. Cifuentes' PhD thesis [14] can be considered the academic foundation for many modern decompilers. The decompilation techniques, based on interval theory and expanded on in follow-up work [15], were implemented in the Intel 80286/DOS to C decompiler `dcc`.

Hex-Rays, a plugin for IDA, is the de facto commercial decompiler. There is no official source on the techniques Hex-Rays uses for decompilation. Schwartz et al. [9] noted that Hex-Rays uses some form of improved structural analysis.

The Phoenix [9] decompiler is built with the Binary Analysis Platform (BAP) [8]. Phoenix attempts to reduce the number of gotos in decompilation through *iterative refinement* which will replace an edge with a goto if the structural analysis algorithm cannot make forward progress. DREAM [44] builds upon the techniques in the Phoenix [9] decompiler to emit zero gotos by design. DREAM forcefully excludes gotos with the use of Boolean expressions and state variables. While it eliminates gotos, it can generate code with entangled execution paths [24]. rev.ng is a decompiler [17, 24] implementing the *Control Flow Combing* algorithm to remove gotos by aggressively duplicating code. These decompilers treat the symptom of the problem—gotos appearing in decompilation—but do not address the root of the problem. SAILR attempts to remedy this by following a simple idea: to achieve high-quality decompilation, decompilers must be compiler-aware.

Recent advances in deep learning techniques have led to progress in the field of decompilation. Researchers have used deep learning approaches to (1) enhance the quality of decompilation by predicting debug information [25], variable names [12, 13, 26, 29], types [12, 46] and function names [6], and (2) build an end-to-end, neural network-based decompiler [23]. These approaches open up exciting opportunities for future research aimed at developing a general-purpose end-to-end neural decompiler.

11 Conclusion

Decompilers seek the impossible—how to recover source code using only binary code. This act of divination requires a complex mix of binary analysis, graph analysis, and, fundamentally, engineering. We believe that the goal of a decompiler should be to get as close as possible to the original code and that the only way to accomplish this is to be compiler-aware. SAILR and the ANGR DECOMPILER represent the next steps in this direction toward perfect decompilation.

Acknowledgement

This project has received funding from the following sources: Defense Advanced Research Projects Agency (DARPA) Contracts No. HR001118C0060, FA875019C0003, N6600120C4020, and N6600122C4026; the Department of the Interior Grant No. D22AP00145-00; the Department of Defense Grant No. H98230-23-C-0270; and National Science Foundation (NSF) Awards No. 2146568 and 2232915.

References

[1] Debian popularity contest statistics: Installations. https://popcon.debian.org/by_inst.

[2] GIMPLE (GNU compiler collection (GCC) internals). https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html.

[3] Hex-Rays – state of the art binary analysis solutions. https://www.hex-rays.com/.

[4] Martin Liska: Switch lowering improvements – slideslive. https://slideslive.com/38902416/switch-lowering-improvements.

[5] Frances E Allen. Control flow analysis. In *Proceedings of the ACM Symposium on Compiler Optimization*, 1970.

[6] Fiorella Artuso, Giuseppe Antonio Di Luna, Luca Massarelli, and Leonardo Querzoni. Function naming in stripped binaries using neural networks. *arXiv preprint arXiv:1912.07946*, 2019.

[7] Martial Bourquin, Andy King, and Edward Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, pages 1–10, 2013.

[8] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[9] 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.

[10] Horst Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern recognition letters*, 18(8):689–694, 1997.

[11] Kevin Burk, Fabio Pagani, Christopher Kruegel, and Giovanni Vigna. Decomperson: How humans decompile and what we can learn from it. In *USENIX Security Symposium*, 2022.

[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 *Proceedings of the USENIX Security Symposium*, August 2022.

[13] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. Varclr: Variable semantic representation pre-training via contrastive learning. *arXiv preprint arXiv:2112.02650*, 2021.

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

[15] Cristina Cifuentes and K John Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995.

[16] John Cocke. Global common subexpression elimination. In *Proceedings of the ACM Symposium on Compiler Optimization*, 1970.

[17] 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.

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

[19] Lukáš 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.

[20] 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 *International Conference on Information Security and Assurance*, pages 72–86. Springer, 2011.

[21] 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.

[22] 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.

[23] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. A neural-based program decompiler. *arXiv:1906.12029 [cs]*, Jun 2019. arXiv: 1906.12029.

[24] 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.

[25] 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.

[26] 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.

[27] Linux Kernel. Linux Kernel, 2023. https://github.com/torvalds/linux/tree/master/kernel.

[28] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[29] 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.

[30] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, page 18, Feb 2011.

[31] 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, Boston, MA, August 2022. USENIX Association.

[32] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. The convergence of source code and binary vulnerability discovery–a case study. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 602–615, 2022.

[33] Omid Mirzaei, Roman Vasilenko, Engin Kirda, Long Lu, and Amin Kharraz. Scrutinizer: Detecting code reuse in malware via decompilation and machine learning. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 130–150. Springer, 2021.

[34] Matthew Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 27–41, Mar 2016.

[35] NSA. The Ghidra decompiler, 2023. https://ghidra-sre.org/.

[36] Pemma Reiter, Hui Jun Tay, Westley Weimer, Adam Doupé, Ruoyu Wang, and Stephanie Forrest. Automatically mitigating vulnerabilities in x86 binary programs via partially recompilable decompilation. *arXiv preprint arXiv:2202.12336*, 2022.

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

[38] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. SoK:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[39] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, Feb 2011.

[40] SpyEye. SpyEye, 2023. https://github.com/ytisf/theZoo/tree/master/malware/Binaries/SpyEye.

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

[42] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 363—-376, New York, NY, USA, 2017. Association for Computing Machinery.

[43] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 158–177. IEEE, 2016.

[44] 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 *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS 2015)*, 2015.

[45] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510, 2013.

[46] 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)*, page 813–832. IEEE, May 2021.

A Appendix

A.1 Popular Debian Packages for Evaluation

We evaluated SAILR on 26 randomly selected from the top 50 Debian packages collected from the Popcon [1] list: bash, libselinux, shadow, libedit, base-passwd, openssh-portable, dpkg, dash, grep, kmod, diffutils, findutils, gnutls, iproute2, gzip, sysvinit, bzip2, libacl, libexpat, libbsd, tar, rsyslog, cronie, zlib, e2fsprogs, and coreutils.

A.2 Case Study: Constant Depropagation

To better understand more complex cases of deoptimization, we document a case study involving constant depropagation in the Coreutils 9.1 package of compounding optimizations.

The binary `fmt` contains an ISD optimization in the function `main`, which causes a duplication of a subgraph shown in Listing 2. However, this case does not match any ISD deoptimization schema because the calls to `xdectoumax` are not the same, differing by a single argument value. The calls on Lines 7 and 11 originate from the same source line and are duplicates, but the constant which is assigned to `max_width` on Line 2 is constant-propagated, causing them to differ after the

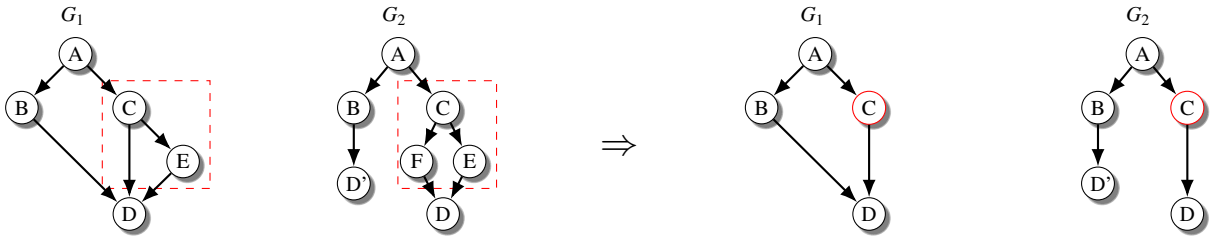

Figure 10: A single round of CFGED approximation and region collapsing. Every round, a region collapses after computing the GED inside the region. Additionally, the difference in the edges source and destination are added as edits.

ISD optimization. To revert this ISD case, the optimizations must be undone in reverse order.

SAILR first runs a KMP similarity matching algorithm to find arguments that may differ across calls. When these arguments are identified, SAILR checks to see if one argument is symbolic (such as a variable) and one is constant. If this is true, SAILR uses Reaching Definitions Analysis to find all the possible values of the symbolic argument on the path with the constant. If the symbolic argument can only equal the constant on that path, it is replaced by the symbolic variable.

In Listing 2 this makes both calls to `xdectoumax` an exact duplicate. Because both calls are duplicates, have a common successor to their blocks, and at least one is connected by a goto, this case matches the ISD algorithm. When this case is merged only one `xdectoumax` remains, with the assignment blocked by the reaching conditions of the two statements.

A.3 Computing CFGED

Figure 10 shows an example computation of CFGED between G_1 and G_2. CFGED starts by finding two tail regions that contain the same head node across graphs. In the decompilation-to-source comparison, the addresses associated with each decompilation line are used to map source lines to nodes. Because a tail region contains no nested regions, the size of these sub-graphs is often small when mapped correctly.

The highlighted regions in Figure 10 have a GED of 2. Because there are also edges leaving these regions, we must check the edit distance of the region's successor. There is a missing edge and node in G1 that should have an edge to the successor. Adding this node and edge constitutes an edit distance of 2. In practice, this can be computed by running GED on the region plus its successor node. In this example, round one has a total CFGED score of 4: 2 for the single edge and node conflict (F, D) and 2 for the GED of the regions.

Continuing, the next region collapsing round would have no more regions to collapse and would run GED on G_1 and G_2. The resulting GED is 3, with the total CFGED to 7, which is equal to the exact GED score of 7.

A.4 Extended Evaluations

Ablation Study. In this experiment, we measure the impact of deoptimization (discussed in Section 5) and knowledge expansion (Section 6). We use three structuring algorithm variants: Phoenix with improvements in Section 6.1, SAILR Lite (Phoenix with improvements in Section 6), and SAILR. We decompile 953 functions in Coreutils 9.1 compiled under optimization level `O2`. As shown in Table 6, both deoptimization and knowledge expansion are necessary for reducing gotos and CFGED scores. Deoptimization contributes more than knowledge expansion in terms of goto reduction.

Linux Kernel Study. To better understand how SAILR structures code highly saturated with gotos, we evaluate SAILR against other structuring algorithms on a subset of the Linux kernel [27]. We randomly picked 100 source files (and object files) from the core kernel code base, collected 802 functions, and ran decompilers against them. Table 7 shows the result: SAILR outperforms other solutions in terms of relative goto emittance to source but shows slightly worse CFGED scores.

```
1  max_width = 0x4b;
2  if (v0) {
3    max_width = xdectoumax(v0, 0x0, 0x9c4, &.LC11,
4        dcgettext(NULL, "invalid width", 0x5), 0x0);
5    if (!v1)
6      goto LABEL_40cd86;
7    goal_width = xdectoumax(v1, 0x0, max_width, &.LC11,
8        dcgettext(NULL, "invalid width", 0x5), 0x0);
9  } else {
10   if (v1) {
11     goal_width = xdectoumax(v1, 0x0, 0x4b, &.LC11,
12         dcgettext(NULL, "invalid width", 0x5), 0x0);
13     max_width = goal_width + 10;
14     goto LABEL_40ccc7;
15   }
16 Label_40cd86:
17   goal_width = (max_width * 187 >> 31
18                 CONCAT max_width * 187) /m 200;
19 }
20 Label_40ccc7:
21 v17 = optind;
```

Listing 2: An example ISD case from the Coreutils binary fmt on function main.

Table 6: Results of the ablation study on Coreutils investigating the impact of deoptimization and knowledge expansion.

	Gotos			Bools			Calls			CFGED		
	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med
Source	40	5	0	1291	27	1.35	11690	173	12.27	0	0	0
Phoenix	1761	37	1.85	534	12	0.56	10136	173	10.64	40733	918	42.74
SAILR Lite	1410	67	1.48	932	20	0.98	10063	173	10.56	40384	928	42.38
SAILR	666	19	0.7	770	20	0.81	10322	173	10.83	40489	1382	42.49

Table 7: Linux kernel structuring evaluation.

	Source	SAILR	Hex-Rays	Ghidra	Phoenix	DREAM	rev.ng
Gotos	120	136	311	352	457	0	0
Bools	737	345	354	519	277	2759	150
Calls	4306	3668	3376	3348	3226	3248	5347
CFGED	0	19096	18916	18576	18956	29400	25903

Table 8: Decompilation of SpyEye with generic readability metrics.

	SAILR	Hex-Rays	Ghidra	Phoenix	DREAM	rev.ng
Gotos	0	0	0	20	0	0
Bools	17	15	17	8	33	1
LoC	1640	878	1881	1690	1681	2186

Malware Study. We also evaluated SAILR on a malware sample: the SpyEye malware on Windows [40]. In Table 8, we compare SAILR against other structuring algorithms on 54 functions. Because there is no source code to compare against, we use generic readability metrics, which may not reflect structural closeness.

Decompilation Times. In Table 9 we show the decompilation times across 1272 functions in Coreutils 9.1 compiled with GCC 9.5 `O2`. The ANGR DECOMPILER takes significantly longer to decompile due to its Python implementation. SAILR takes longer than our other structuring algorithms because of a naive implementation for the search phase for ISD deoptimization. In short, SAILR uses a shortest-path algorithm to find the head for deduplication, which we believe can eliminated with future engineering.

Table 9: Decompilation times (in seconds) across all functions in Coreutils.

	SAILR	Hex-Rays	Ghidra	Phoenix	DREAM	rev.ng
Median	1.16	0.01	0.05	0.82	0.8	0.71
Mean	5.24	0.03	0.07	2.15	2.08	2.33
Max	606.7	0.38	1.06	55.32	31.55	329.33
Sum	6670.0	31.84	91.43	2735.03	2647.14	2962.65

Compiler Configurations Study. To understand the full effect of different compiler optimizations on SAILR, we compiled the same 26 Debian packages in Section 8.3 on all optimization levels for GCC 9.5. In Table 10 we show a comparison against other structuring algorithms on 5,752 functions. The result shows that SAILR consistently yields the closest number of gotos (except `O0`) with CFGED scores being very close (within 6%) to the lowest CFGED scores (from Hex-Rays). We believe SAILR's poor results on gotos and CFGED compared to Hex-Rays is due to Hex-Rays maturity in schema for unoptimized binaries.

Expanded Multi-Compiler Study. We expanded our results from Table 4 in Table 11 to include all decompilers supported in our evaluation pipeline. Note, that there are fewer functions (392 total) in these results due to functions missing across more decompilers.

```
def deoptimize_isd(duplicate_graph_pair, graph):
    merged_graph = merge_pair_graphs(duplicate_graph_pair)
    remove_pair_nodes_from_graph(graph_pair, graph)
    graph = compose_graphs_of_many(graph, merged_graph)
    for head_node in heads_of_graphs(merged_graph):
        point_old_predecessors_to_node(head_head)
    for leaf_node in ends_of_graphs(merged_graph):
        conditional_node = make_reaching_cond_node(leaf_node)
        graph.add_edge(leaf_node, conditional_node)
    for node in new_conditional_nodes(graph):
        original_leafs = leaf_nodes(duplicate_graph_pair)
        for leaf in original_leafs:
            graph.add_edge(
                node,
                original_edge_by_condition(leaf)
            )
    for node in new_conditional_nodes(graph):
        simplify_and_merge_adjacent_conditionals(node)
    return graph
```

Listing 3: Pseudocode of the SAILR ISD deoptimization algorithm. It takes as input a pair of correctly identified duplicate graph pairs and the original graph to be deoptimized. It outputs the deoptimized graph.

```
def deoptimize_isc(target_node, graph):
    exit_regions = get_exit_regions(graph)
    node_chain = []
    next_node = graph.successors(target_node)[0]
    graph.remove_edge(target_node, next_node)
    while next_node is not None:
        node_chain.append(next_node)
        successors = graph.successors(next_node)
        if len(successors) > 1:
            break
        next_node = successors[0]
    last_node = target_node
    for node in node_chain
        if is_region_head(node, exit_regions):
            new_nodes = copy_region_nodes(node)
            graph.add_edges(last_node, new_nodes)
            break
        else:
            graph.add_edge(last_node, copy(node))
            last_node = node
    return graph
```

Listing 4: Pseudocode of the SAILR ISC deoptimization algorithm.

Table 10: A comparison of structuring algorithms across optimization levels on GCC 9.5 for Debian packages.

	Metric	O0			O1			O2			O3		
		Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med
Source	Gotos	1000	0.17	0.0	1000	0.17	0.0	1000	0.17	0.0	1000	0.17	0.0
	Bools	4252	0.74	0.0	4256	0.74	0.0	4256	0.74	0.0	4256	0.74	0.0
	Calls	38828	6.75	3.0	38897	6.76	3.0	38897	6.76	3.0	38897	6.76	3.0
	CFGED	0	0	0.0	0	0	0.0	0	0	0.0	0	0	0.0
SAILR	Gotos	426	0.07	0.0	1148	0.2	0.0	1542	0.27	0.0	1596	0.28	0.0
	Bools	3382	0.59	0.0	2628	0.46	0.0	2651	0.46	0.0	2731	0.47	0.0
	Calls	38021	6.61	3.0	38818	6.75	3.0	37921	6.59	3.0	38221	6.64	3.0
	CFGED	92198	16.03	5.0	105786	18.39	7.0	110358	19.19	7.0	115517	20.08	7.0
Hex-Rays	Gotos	652	0.11	0.0	2654	0.46	0.0	3718	0.65	0.0	3924	0.68	0.0
	Bools	4416	0.77	0.0	2935	0.51	0.0	2936	0.51	0.0	3046	0.53	0.0
	Calls	39433	6.86	3.0	39250	6.82	3.0	38037	6.61	3.0	38311	6.66	3.0
	CFGED	86658	15.07	4.0	104118	18.1	7.0	109028	18.95	7.0	114335	19.88	7.0
Ghidra	Gotos	1453	0.25	0.0	3311	0.58	0.0	4209	0.73	0.0	4324	0.75	0.0
	Bools	4341	0.75	0.0	3270	0.57	0.0	3366	0.59	0.0	3741	0.65	0.0
	Calls	39305	6.83	3.0	40203	6.99	3.0	38767	6.74	3.0	38960	6.77	3.0
	CFGED	117150	20.37	5.0	123810	21.52	7.0	125989	21.9	7.0	130729	22.73	7.0
Phoenix	Gotos	3767	0.65	0.0	4438	0.77	0.0	5501	0.96	0.0	5712	0.99	0.0
	Bools	2631	0.46	0.0	1803	0.31	0.0	1878	0.33	0.0	1960	0.34	0.0
	Calls	38034	6.61	3.0	38000	6.61	3.0	37062	6.44	3.0	37303	6.49	3.0
	CFGED	94257	16.39	4.0	106807	18.57	7.0	110871	19.28	7.0	115350	20.05	7.0
DREAM	Gotos	0	0	0	0	0	0	0	0	0	0	0	0
	Bools	10976	1.91	0.0	17448	3.03	0.0	28654	4.98	0.0	30048	5.22	0.0
	Calls	38036	6.61	3.0	38023	6.61	3.0	37090	6.45	3.0	37337	6.49	3.0
	CFGED	138318	24.05	5.0	176340	30.66	8.0	223428	38.84	9.0	234183	40.71	9.0
rev.ng	Gotos	0	0	0	0	0	0	0	0	0	0	0	0
	Bools	1732	0.3	0.0	1747	0.3	0.0	960	0.17	0.0	974	0.17	0.0
	Calls	93713	16.29	3.0	108128	18.8	3.0	97076	16.88	3.0	97752	16.99	3.0
	CFGED	271168	47.14	6.0	292823	50.91	7.0	282414	49.1	7.0	295824	51.43	7.0

Table 11: A comparison of decompilation on Coreutils binaries compiled by GCC 5, GCC 9, GCC 11, and Clang 14 with O2.

	Metric	GCC 5			GCC 9			GCC 11			Clang 14		
		Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med	Sum	Avg	Med
Source	Gotos	4	0.01	0.0	4	0.01	0.0	4	0.01	0.0	4	0.01	0.0
	Bools	273	0.7	0.0	263	0.67	0.0	263	0.67	0.0	266	0.68	0.0
	Calls	3724	9.5	5.0	3726	9.51	5.0	3726	9.51	5.0	3781	9.65	5.0
	CFGED	0	0	0	0	0	0	0	0	0	0	0	0
SAILR	Gotos	56	0.14	0.0	68	0.17	0.0	70	0.18	0.0	66	0.17	0.0
	Bools	168	0.43	0.0	159	0.41	0.0	160	0.41	0.0	172	0.44	0.0
	Calls	3418	8.72	4.0	3430	8.75	4.0	3420	8.72	4.0	3431	8.75	4.0
	CFGED	11880	30.31	7.5	8209	20.94	8.0	8272	21.1	7.5	9027	23.03	9.0
Hex-Rays	Gotos	208	0.53	0.0	226	0.58	0.0	224	0.57	0.0	216	0.55	0.0
	Bools	184	0.47	0.0	179	0.46	0.0	177	0.45	0.0	180	0.46	0.0
	Calls	3485	8.89	5.0	3485	8.89	5.0	3487	8.9	5.0	3405	8.69	4.0
	CFGED	11727	29.92	7.0	8065	20.57	7.0	8041	20.51	7.0	8966	22.87	9.0
Ghidra	Gotos	216	0.55	0.0	215	0.55	0.0	203	0.52	0.0	101	0.26	0.0
	Bools	208	0.53	0.0	203	0.52	0.0	206	0.53	0.0	228	0.58	0.0
	Calls	3525	8.99	5.0	3579	9.13	5.0	3577	9.12	5.0	1846	4.71	2.0
	CFGED	12373	31.56	7.0	8878	22.65	7.5	8810	22.47	7.0	8859	22.6	7.0
Phoenix	Gotos	291	0.74	0.0	306	0.78	0.0	300	0.77	0.0	339	0.86	0.0
	Bools	111	0.28	0.0	115	0.29	0.0	113	0.29	0.0	115	0.29	0.0
	Calls	3397	8.67	4.0	3399	8.67	4.0	3400	8.67	4.0	3403	8.68	4.0
	CFGED	11966	30.53	7.0	8370	21.35	8.0	8434	21.52	7.5	9357	23.87	9.0
DREAM	Gotos	0	0	0.0	0	0	0.0	0	0	0.0	0	0	0.0
	Bools	1569	4.0	0.0	1640	4.18	0.0	1774	4.53	0.0	1411	3.6	0.0
	Calls	3397	8.67	4.0	3400	8.67	4.0	3401	8.68	4.0	3402	8.68	4.0
	CFGED	16940	43.21	9.0	13476	34.38	9.5	13389	34.16	9.0	14245	36.34	12.0
rev.ng	Gotos	0	0	0.0	0	0	0.0	0	0	0.0	0	0	0.0
	Bools	29	0.07	0.0	33	0.08	0.0	32	0.08	0.0	72	0.18	0.0
	Calls	5228	13.34	5.0	5284	13.48	5.0	5235	13.35	5.0	5738	14.64	5.0
	CFGED	17457	44.53	7.0	13289	33.9	8.0	13447	34.3	8.0	17066	43.54	10