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

USENIX ’24
GOTO instruction

I could restructure the program’s flow or use one little ‘GOTO’ instead.

Eh, screw good practice. How bad can it be?

```
goto main_sub3;
*compile*
```
Good Binary Decompilation = No GOTO?

How should we interpret GOTO statements in decompiled code?
What is wrong with goto?

- Structuring failures manifest as goto statements in the decompilation

- However, in Linux kernel (6.1): 3,754 gotos exist
  - Having no goto = better decompilation?
Success of a decompiler = Eliminate all gotos?

```c
int schedule_job(int needs_next, int fast_job, int mode)
{
    if (needs_next && fast_job) {
        complete_job();
        if (mode == EARLY_EXIT)
            goto cleanup;

        next_job();
    }

    refresh_jobs();
    if (fast_job)
        fast_unlock();

cleanup:
    complete_job();
    log_workers();
    return job_status(fast_job);
}
```

Listing 1: A motivating example based on code from the Linux kernel job scheduler.
Which Decompilation is better?

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

• There are developer intended gotos

• Decompilation should aim to be as close to original source code

• Decompilation should preserve intended gotos and eliminate unintended (spurious) gotos
Q1. What causes spurious gotos?

Q2. How can we preserve the intended structure?

Q3. How good is the new structuring algorithm compared to previous work?
A1. Cause of Spurious GOTOs
Searching for Spurious GOTO

• Manual Search -> (Does not Scale!)

• 1. Compile binary using GCC (O2, save-temps, dump-tree-all)
  • Saves intermediate files

• 2. Decompile all functions -> identify functions /w GOTO but /wo GOTO in source
Unstructurable Subgraph

• Structuring Algorithms attempt to match the subgraph of a CFG against known control-flow patterns for C control-flow structures

• Unstructurable subgraph = does not match a known C control-flow pattern.
  • Compiler optimizations create novel graph schemas
Optimizations that Cause Spurious GOTO? (Coreutils 9.1, GCC O2 -> O0)

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.
A. Jump Threading

• Transforms a conditional branch into an unconditional branch for certain paths
B. Common Subexpression Elimination

- When a common statement is shared among multiple blocks
  - Reduce to one expression and jump to that expression
D. Cross Jumping

- Unifies duplicate code and replaces duplicates with a jump to a unified statement.
Others

• Switch Conversion (C)
• Software Thread Cache Reordering (E)
• Loop Header Optimization (F)
• Built-in Inlining (G)
• Switch Lowering (H)
• Nonreturning Functions (I)
Classification

• Irreducible Statement Duplications (ISD)  
  (Single -> Many)  
  • A, E, F  
• Irreducible Statement Condensing (ISC)  
  (Many->Single)  
  • B, C, D  
• Miscellaneous  
  • G, H, I
A2. SAILR: A Compiler-Aware Structuring Algorithm
Figure 3: Overview of ANGR DECOMPILER’s decompilation pipeline.
Angr Decompiler

• Binary -> VEX IR CFG -> AIL CFG (Angr Intermediate Language CFG)

• SAILR:
  • AIL CFG (IN), C Pseudocode (OUT)

  • Region Identification
  • Schema Matching
  • Region Simplifications
  • Deoptimization (*)
    • How? “Compiler-Aware”
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.
Deoptimizing ISD

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.
ISC Optimization

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

```c
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.
Deoptimizing ISC

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.
A3. Evaluation
Measuring the Quality of Structuring

• Number of gotos (GOTos) (prev)
• McCabe Cyclomatic Code Complexity (MCC) (prev)
• Line of Code (LoC) (prev)
• Graph Edit Distance (GED) (new)
• Control-Flow Graph Edit Distance (CFGED) (new)
Figure 1: (From left to right) the DREAM, Phoenix, and SAILR decompilation of Listing 1 (using GCC 9.5 -O2).
Measuring the Quality of Structuring

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

<table>
<thead>
<tr>
<th></th>
<th>Gotos</th>
<th>LoC</th>
<th>MCC</th>
<th>GED</th>
<th>CFGED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>1</td>
<td>19</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SAILR</td>
<td>1</td>
<td>15</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Phoenix</td>
<td>3</td>
<td>19</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>DREAM</td>
<td>0</td>
<td>16</td>
<td>9</td>
<td>21</td>
<td>38</td>
</tr>
</tbody>
</table>
GED and CFGED

• GED
  • Edge-node location difference metric between source CFG and decompiled code CFG

• CFGED
  • GED is usually too expensive to compute on a graph with >12 nodes
  • Identify Single-entry Single-exit (SESE) regions and compute GED for each region (CFGED = sum of GED of all SESE regions)
  • Approximate of exact GED
### Evaluation

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.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Source</th>
<th>SAIL.R</th>
<th>Hex-Rays</th>
<th>Ghidra</th>
<th>Phoenix</th>
<th>DREAM</th>
<th>rev.ng</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gotos</td>
<td>1,367</td>
<td>0.19</td>
<td>0.0</td>
<td>2,673</td>
<td>(95.5%)</td>
<td>0.36</td>
<td>0.0</td>
</tr>
<tr>
<td>Bools</td>
<td>6,180</td>
<td>0.84</td>
<td>0.0</td>
<td>3,980</td>
<td>(35.6%)</td>
<td>0.54</td>
<td>0.0</td>
</tr>
<tr>
<td>Calls</td>
<td>53,995</td>
<td>7.34</td>
<td>3.0</td>
<td>52,558</td>
<td>(2.6%)</td>
<td>7.15</td>
<td>3.0</td>
</tr>
<tr>
<td>CFGED</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>106,468</td>
<td>(0.5%)</td>
<td>2.04</td>
<td>8.0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Source</th>
<th>Most-Recent Release</th>
<th>Decompiler</th>
<th>Gotos</th>
<th>Bools</th>
<th>Calls</th>
<th>CFGED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>N/A</td>
<td>SAIL.R</td>
<td>438</td>
<td>4,761</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>GCC 5</td>
<td>October 10, 2017</td>
<td>SAIL.R</td>
<td>152</td>
<td>295</td>
<td>4260</td>
<td>14499</td>
</tr>
<tr>
<td>GCC 9</td>
<td>May 27, 2022</td>
<td>SAIL.R</td>
<td>169</td>
<td>284</td>
<td>4277</td>
<td>13462</td>
</tr>
<tr>
<td>GCC 11</td>
<td>April 21, 2022</td>
<td>SAIL.R</td>
<td>170</td>
<td>280</td>
<td>4290</td>
<td>13445</td>
</tr>
<tr>
<td>Clang 14</td>
<td>March 25, 2022</td>
<td>SAIL.R</td>
<td>167</td>
<td>292</td>
<td>4290</td>
<td>22879</td>
</tr>
<tr>
<td></td>
<td>Clang 14</td>
<td>Hex-Rays</td>
<td>454</td>
<td>303</td>
<td>4353</td>
<td>22671</td>
</tr>
</tbody>
</table>
Why is CFGED so large?

• CFGED is still an approximation
• For large CFGs, CFGED differ significantly from GED
  • When exact GED is 4, CFGED reported 306
Takeaway

• Identifies the root cause of spurious gotos in disassembly (compiler optimizations)

• Proposes a decompilation evaluation metric

• Highlights the importance of binary provenance information for decompilation