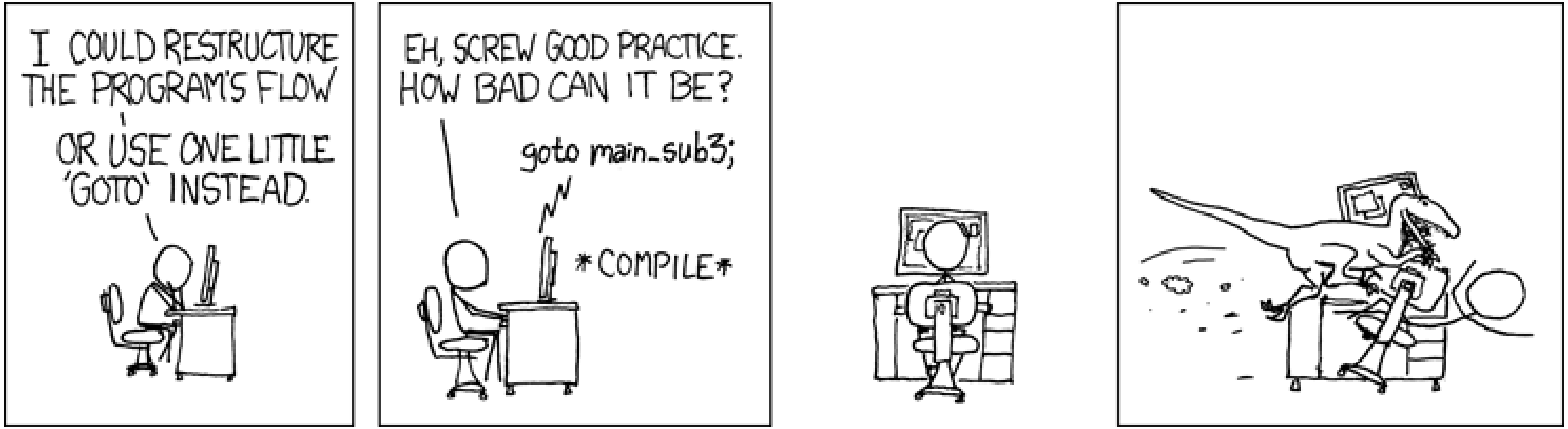
Ahoy SAILR! There is No Need to DREAM of C: A Compiler-Aware Structuring Algorithm for Binary Decompilation

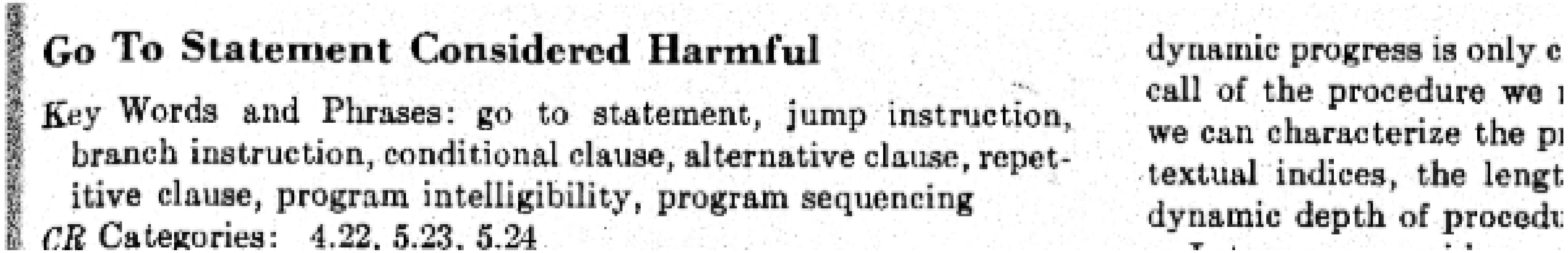
Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang, Arizona State University

USENIX ‘24

GOTO instruction



Edgar Dijkstra: Go To Statement Considered Harmful



Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

CR Categories: 4.22, 5.23, 5.24

dynamic progress is only c
call of the procedure we
we can characterize the p
textual indices, the lengt
dynamic depth of proced

Good Binary Decompilation = No GOTO?

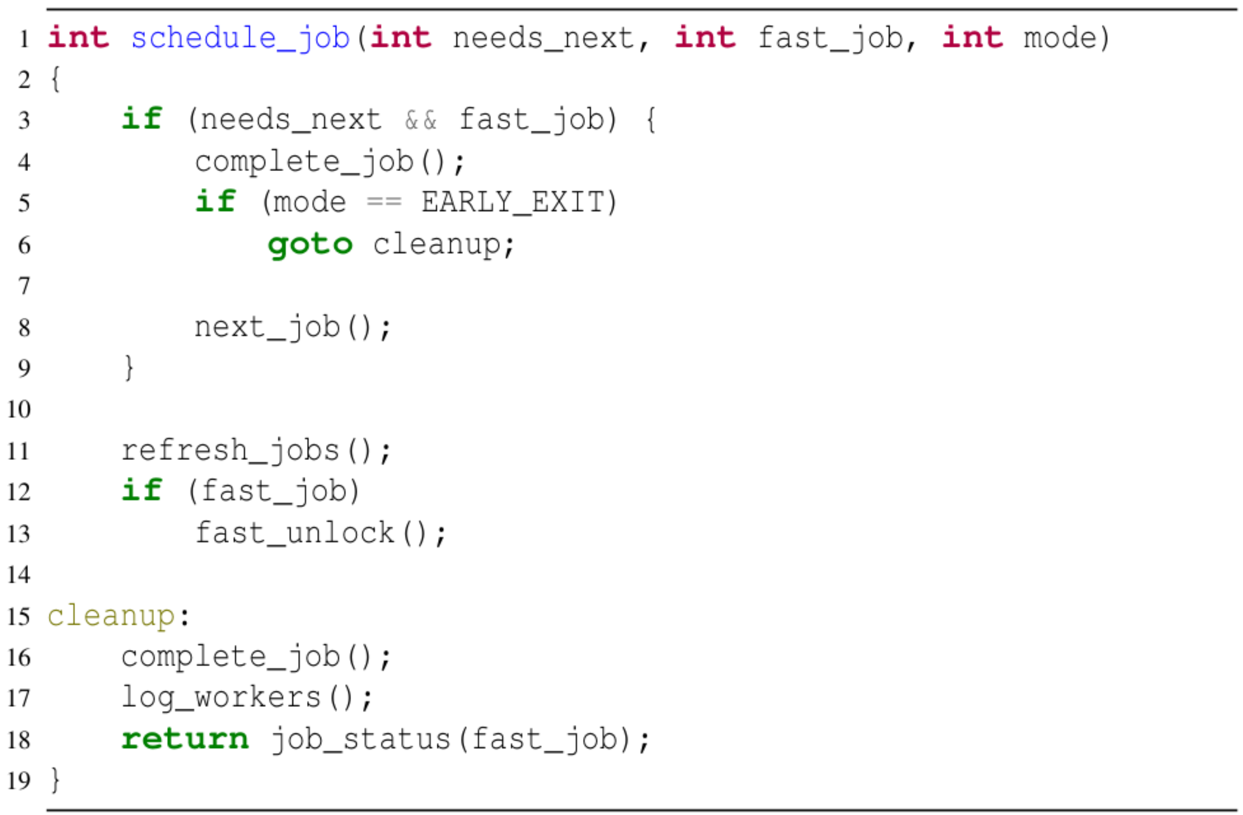
How should we interpret GOTO statements in decompiled code?

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?*

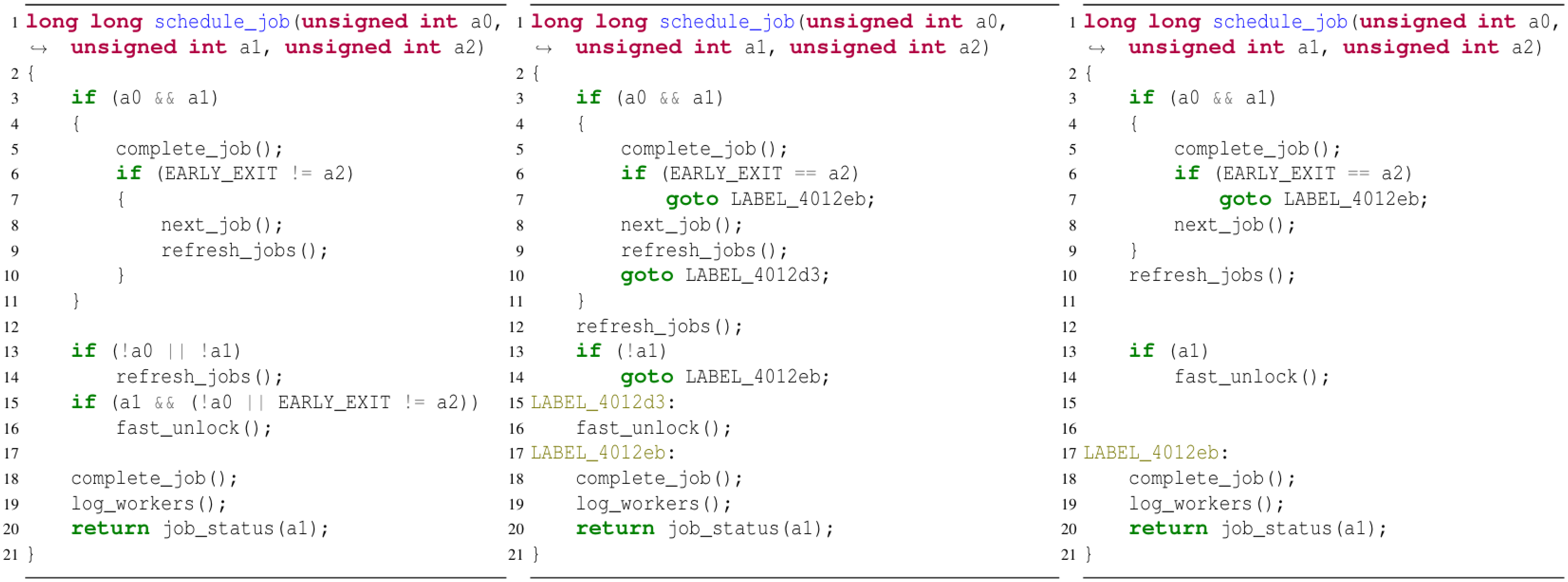
Success of a decompiler = Eliminate all gotos?



```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?



```c
long long schedule_job(unsigned int a0, unsigned int a1, unsigned int a2)
{
    if (a0 && a1)
    {
        complete_job();
        if (EARLY_EXIT != a2)
        {
            next_job();
            refresh_jobs();
        }
    }

    if (!a0 || !a1)
        refresh_jobs();
    if (a1 && (!a0 || EARLY_EXIT != a2))
        fast_unlock();

    complete_job();
    log_workers();
    return job_status(a1);
}
```

```c
long long schedule_job(unsigned int a0, unsigned int a1, unsigned int a2)
{
    if (a0 && a1)
    {
        complete_job();
        if (EARLY_EXIT == a2)
            goto LABEL_4012eb;
        next_job();
        refresh_jobs();
        goto LABEL_4012d3;
    }
    refresh_jobs();
    if (!a1)
        goto LABEL_4012eb;
LABEL_4012d3:
    fast_unlock();
LABEL_4012eb:
    complete_job();
    log_workers();
    return job_status(a1);
}
```

```c
long long schedule_job(unsigned int a0, unsigned int a1, unsigned int a2)
{
    if (a0 && a1)
    {
        complete_job();
        if (EARLY_EXIT == a2)
            goto LABEL_4012eb;
        next_job();
    }
    refresh_jobs();


    if (a1)
        fast_unlock();


LABEL_4012eb:
    complete_job();
    log_workers();
    return job_status(a1);
}
```

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!)

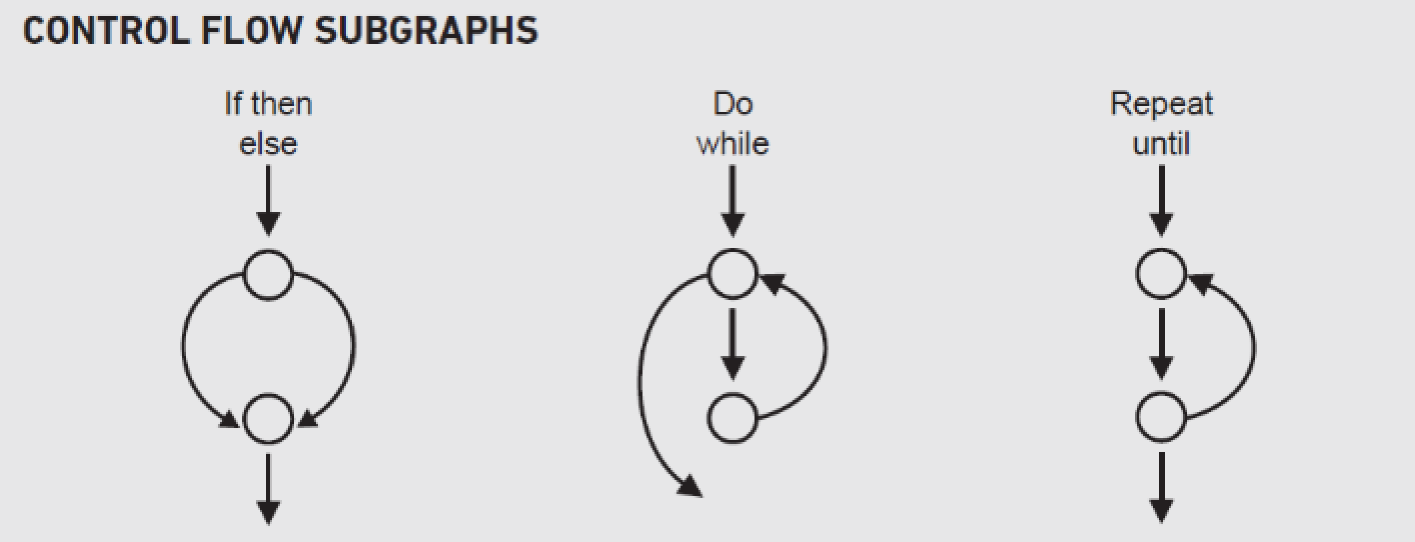
- 1. Compile binary using GCC (O2, save-temps, dump-tree-all)
  - Saves intermediate files

- 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)

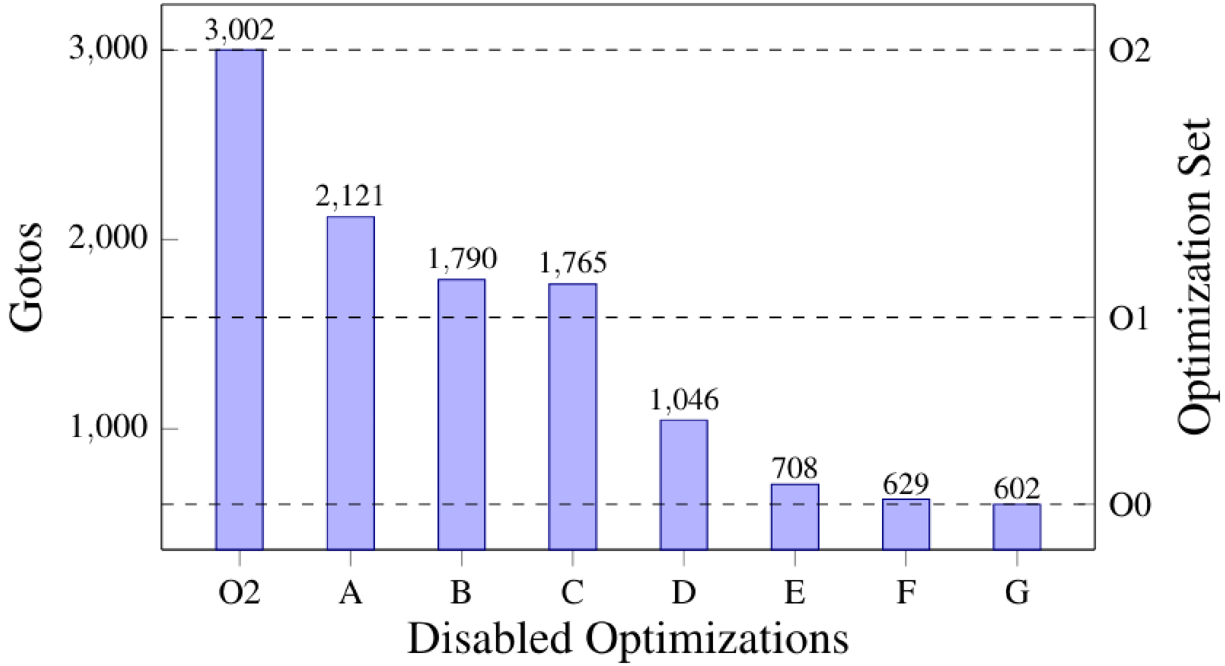
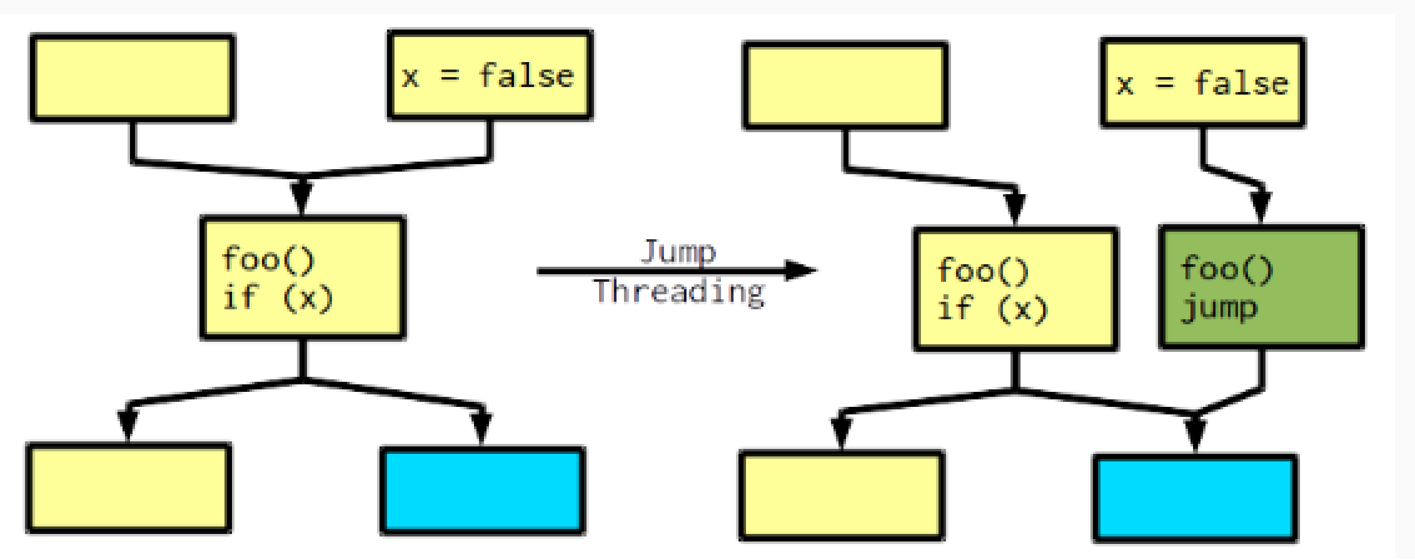


Figure 2: Gotos present in Hex-Rays decompilation as optimizations in Section 3.3 are disabled. Each optimization point disables itself and all optimizations to its left. Optimization sets `O2` through `O0` are shown for reference.

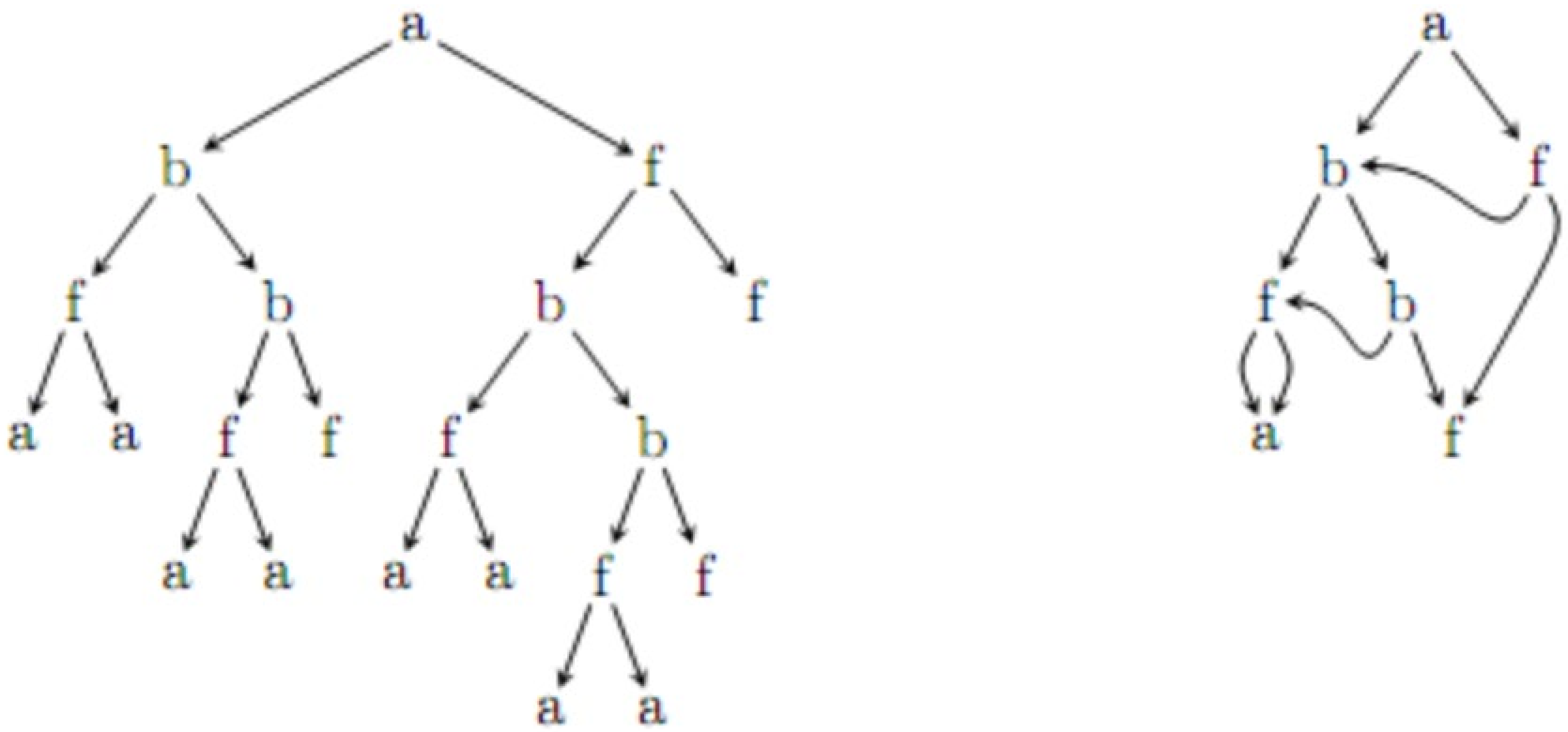
A. Jump Threading

- Transforms a conditional branch into an unconditional branch for certain paths



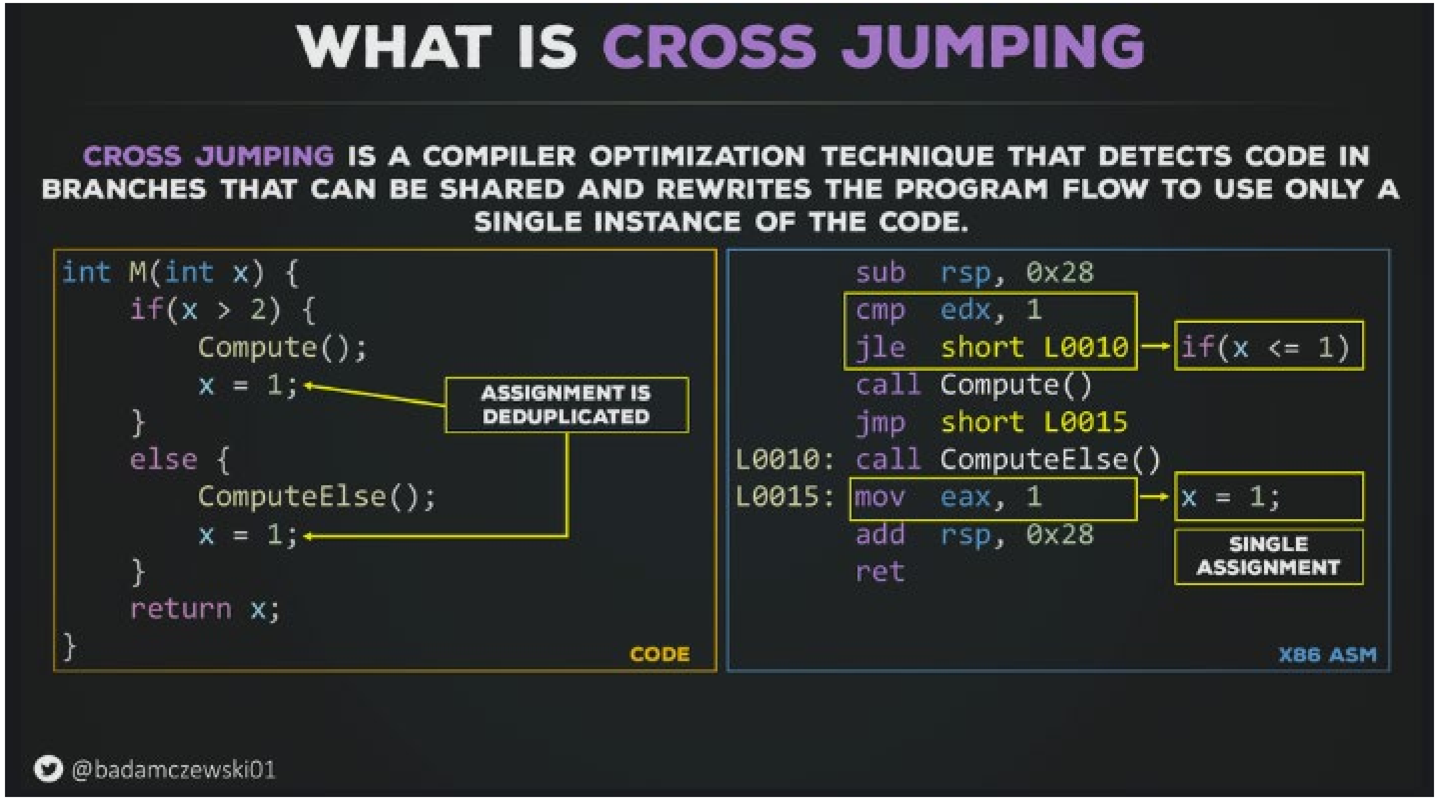
B. Common Subexpression Elimination

- 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 replace duplicates with a jump to unified statement



WHAT IS CROSS JUMPING

CROSS JUMPING IS A COMPILER OPTIMIZATION TECHNIQUE THAT DETECTS CODE IN BRANCHES THAT CAN BE SHARED AND REWRITES THE PROGRAM FLOW TO USE ONLY A SINGLE INSTANCE OF THE CODE.

```
int M(int x) {
    if(x > 2) {
        Compute();
        x = 1;
    }
    else {
        ComputeElse();
        x = 1;
    }
    return x;
}
```

```
        sub  rsp, 0x28
        cmp  edx, 1
        jle  short L0010
        call Compute()
        jmp  short L0015
L0010:  call ComputeElse()
L0015:  mov  eax, 1
        add  rsp, 0x28
        ret
```

Others

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

Classification

- Irreducible Statement Duplications (ISD)

(Single -> Many)

  - A, E, F

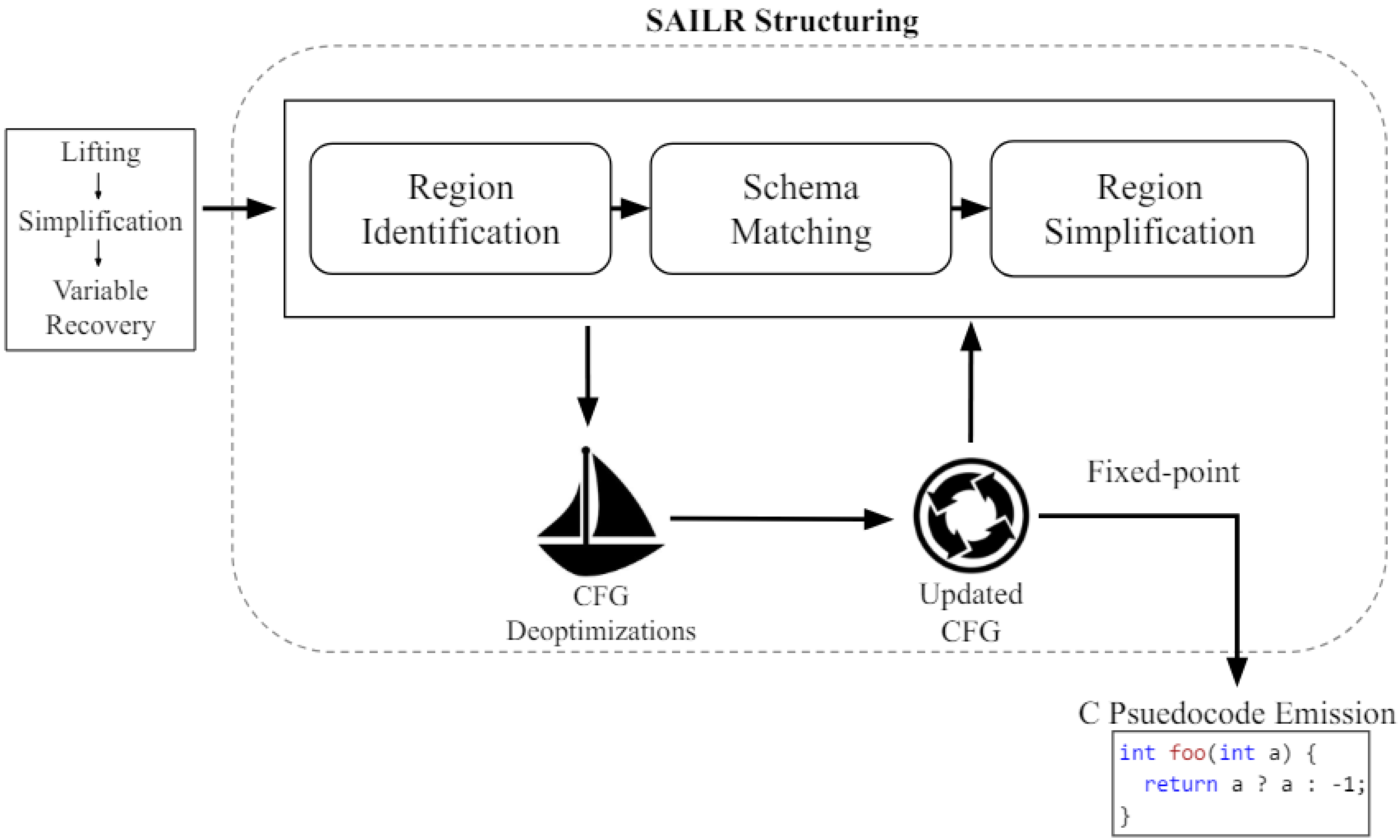
- Irreducible Statement Condensing (ISC)

(Many->Single)

  - B, C, D

- Miscellaneous
  - G, H, I

A2. SAILR: A Compiler-Aware Structuring Algorithm



```
int foo(int a) {
  return a ? a : -1;
}
```

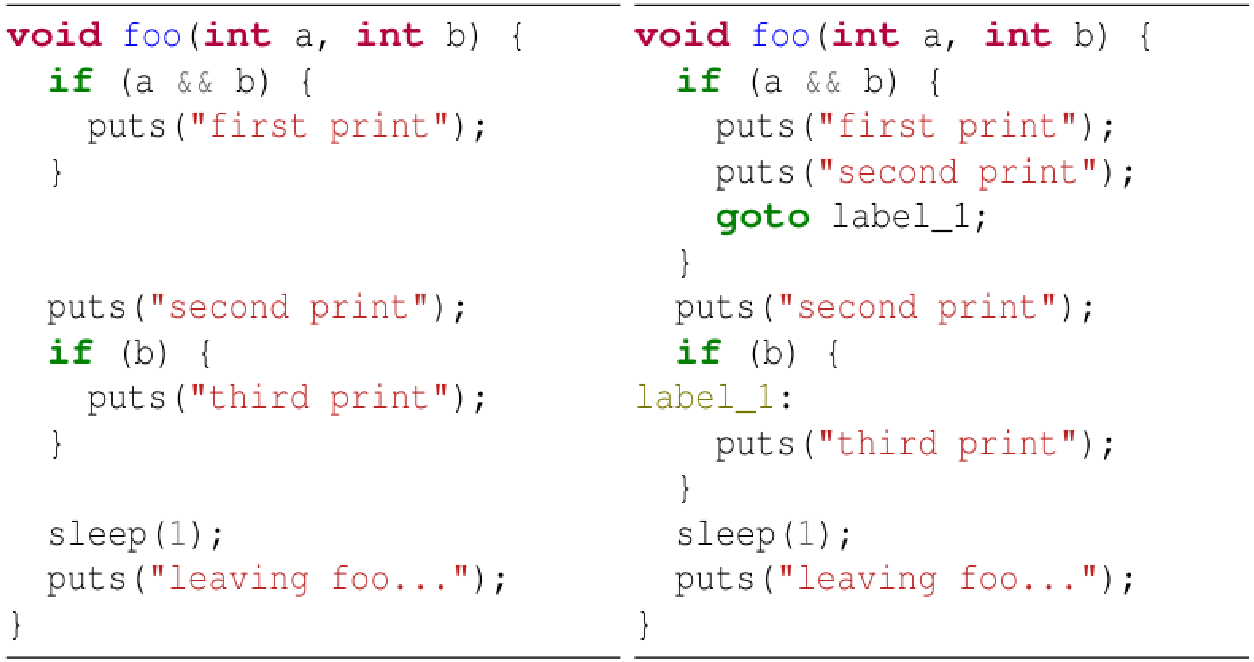
Figure 3: Overview of ANGR DECOMPILER's decompilation pipeline.

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”

ISD Optimizations



```
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.

Deoptimizing ISD

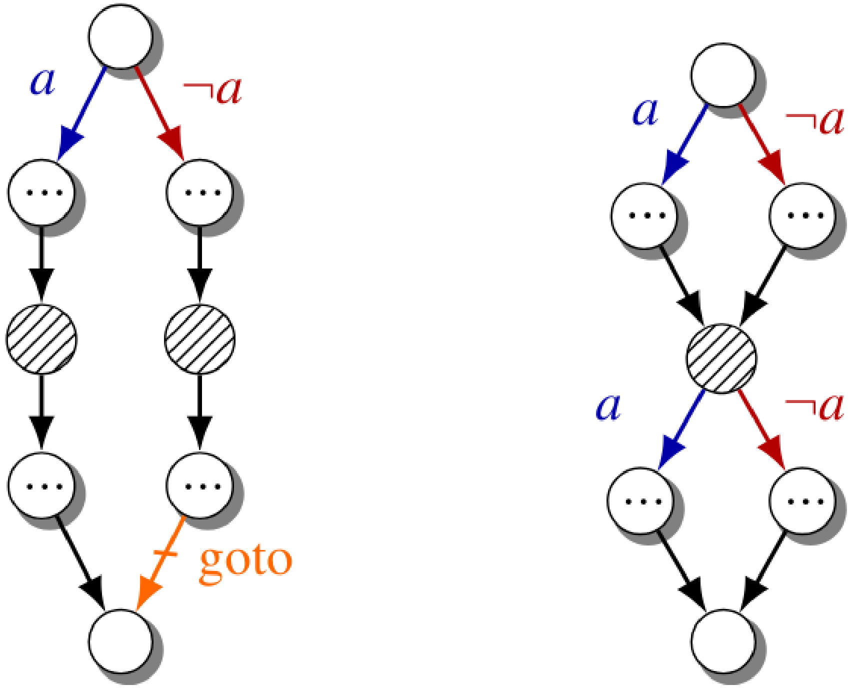
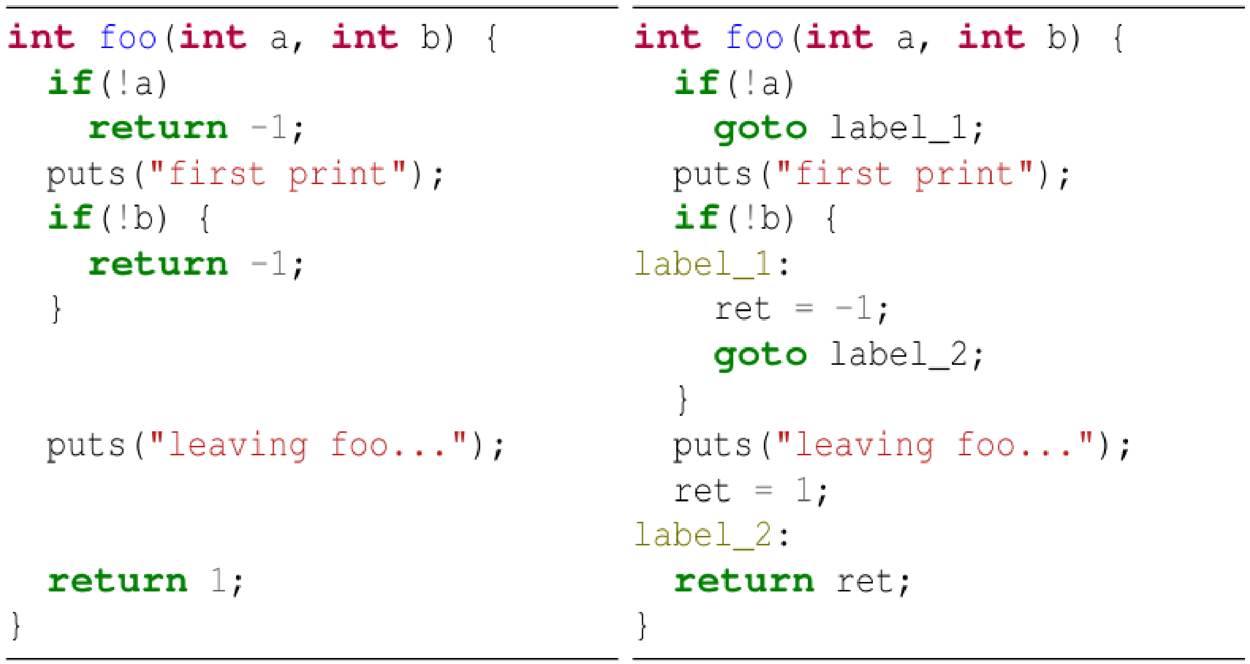


Figure 5: CFGs before and after deoptimizing an ISD optimization case. In order to identify an ISD case, the shaded node must be found to have a semantic duplicate, as well as post-dominating goto edge. The nodes are merged and then bounded by their previous conditions.

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

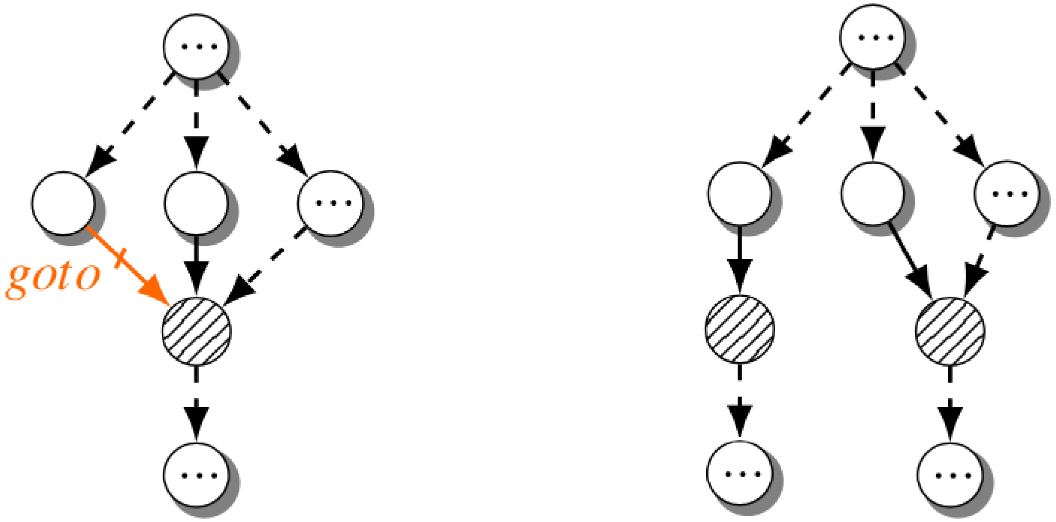


Figure 6: CFGs before and after deoptimizing an ISC optimization case. ISC cases contain a goto edge connecting one node to another node that has multiple predecessors. Duplication of the shaded node, and all its single-successor ancestors, revert this case.

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)

```c
long long schedule_job(unsigned int a0, unsigned int a1, unsigned int a2)
{
    if (a0 && a1)
    {
        complete_job();
        if (EARLY_EXIT != a2)
        {
            next_job();
            refresh_jobs();
        }
    }

    if (!a0 || !a1)
        refresh_jobs();
    if (a1 && (!a0 || EARLY_EXIT != a2))
        fast_unlock();

    complete_job();
    log_workers();
    return job_status(a1);
}
```

```c
long long schedule_job(unsigned int a0, unsigned int a1, unsigned int a2)
{
    if (a0 && a1)
    {
        complete_job();
        if (EARLY_EXIT == a2)
            goto LABEL_4012eb;
        next_job();
        refresh_jobs();
        goto LABEL_4012d3;
    }
    refresh_jobs();
    if (!a1)
        goto LABEL_4012eb;
LABEL_4012d3:
    fast_unlock();
LABEL_4012eb:
    complete_job();
    log_workers();
    return job_status(a1);
}
```

```c
long long schedule_job(unsigned int a0, unsigned int a1, unsigned int a2)
{
    if (a0 && a1)
    {
        complete_job();
        if (EARLY_EXIT == a2)
            goto LABEL_4012eb;
        next_job();
    }
    refresh_jobs();


    if (a1)
        fast_unlock();


LABEL_4012eb:
    complete_job();
    log_workers();
    return job_status(a1);
}
```

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

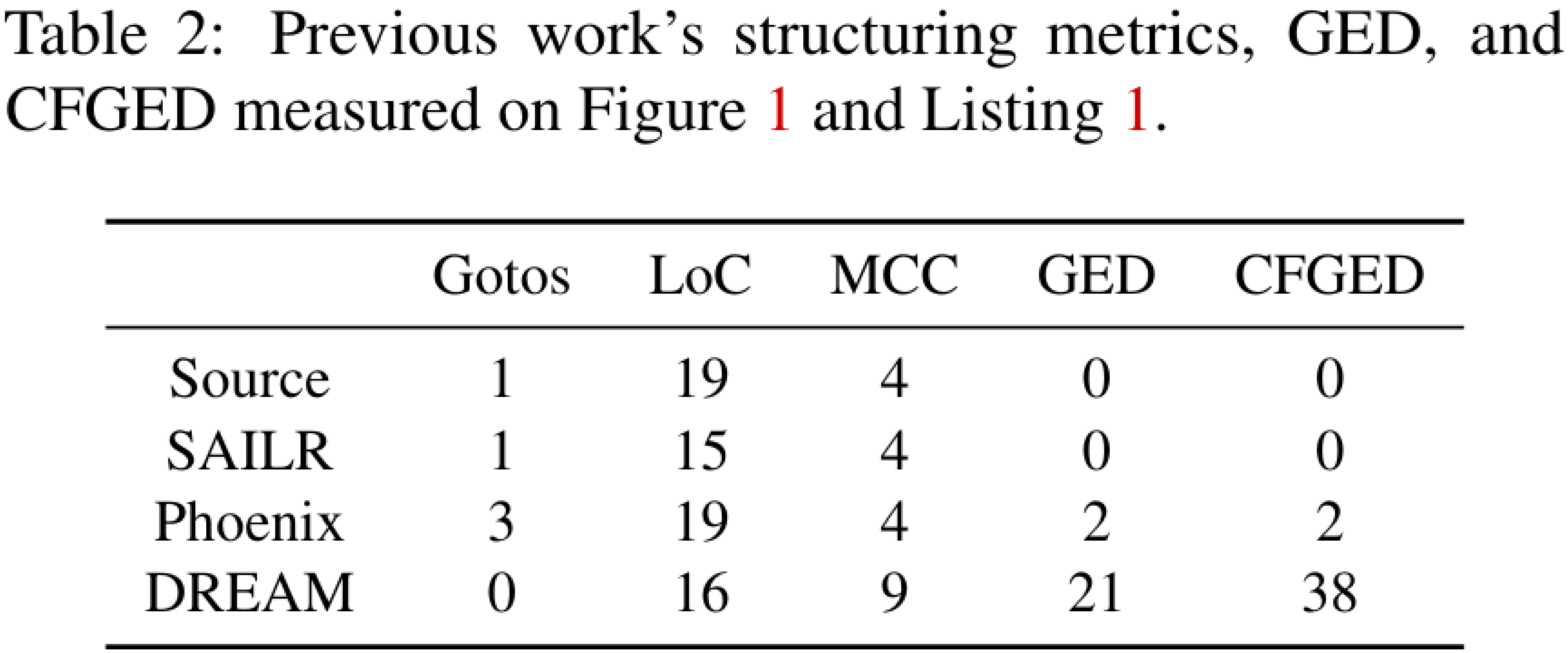


Table 2: Previous work's structuring metrics, GED, and CFGED measured on Figure 1 and Listing 1.

	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

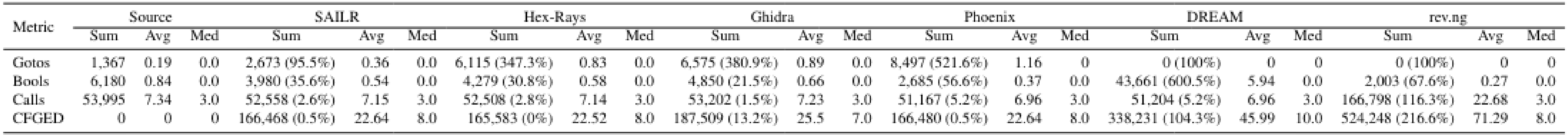
GED and CFGED

- 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

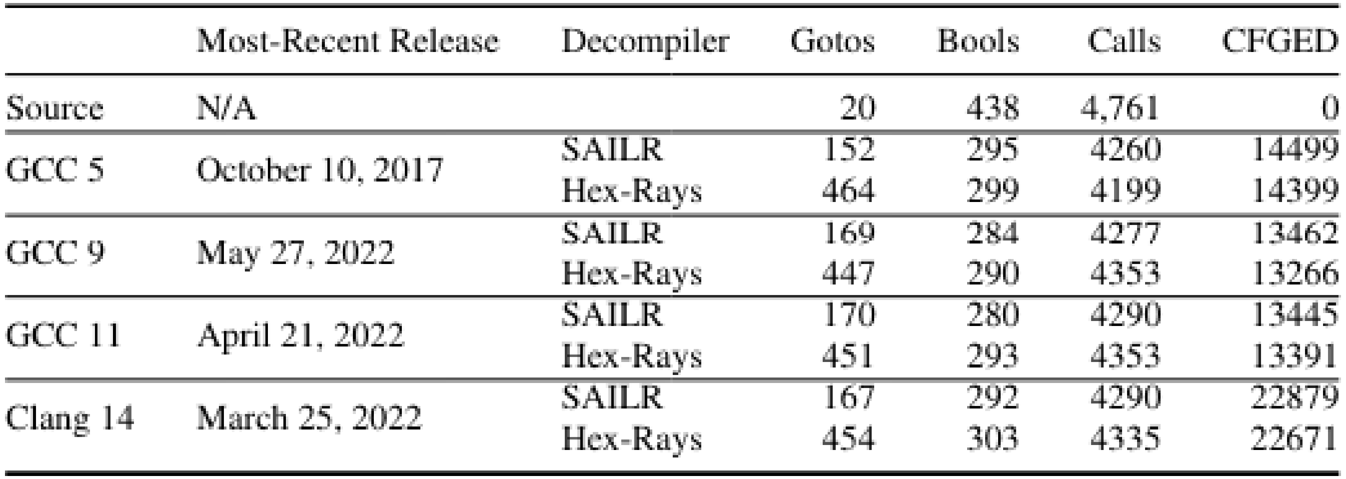
Evaluation

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.



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

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

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