Control Flow Structuring in Radeco

Michael Zhang

About Me

- HMPerson1 on GitHub and Telegram
- 2nd year CS major at Purdue University
- Uses r2 to play CTFs
- Likes compilers

Radeco Design Recap

1. ESIL -> Radeco IR
2. Radeco IR -> C-CFG
3. C-CFG -> C-AST

No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations

Khaled Yakdan*, Sebastian Eschweiler†, Elmar Gerhards-Padilla†, Matthew Smith*
*University of Bonn, Germany
{*yakdan, smith*}*@cs.uni-bonn.de*
†Fraunhofer FKIE, Germany
{*sebastian.eschweiler, elmar.gerhards-padilla*}*@fkie.fraunhofer.de*

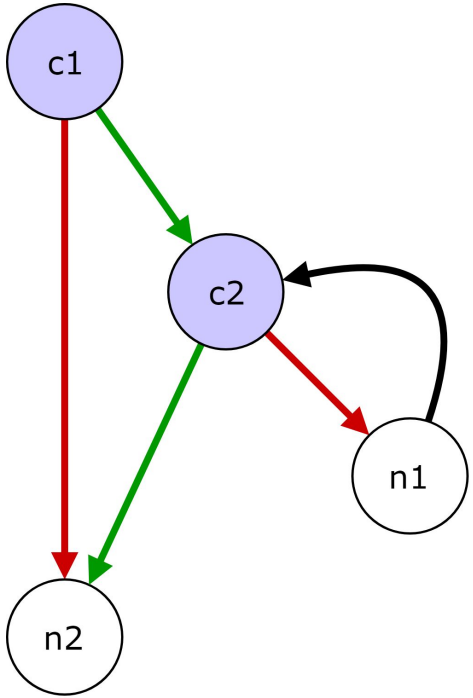
***Abstract*—Decompilation is important for many security applications; it facilitates the tedious task of manual malware reverse engineering and enables the use of source-based security tools on binary code. This includes tools to find vulnerabilities, discover bugs, and perform taint tracking. Recovering high-level control constructs is essential for decompilation in order to produce structured code that is suitable for human analysts and source-based program analysis techniques. State-of-the-art decompilers rely on structural analysis, a pattern-matching approach over the control flow graph, to recover control constructs from binary code. Whenever no match is found, they generate `goto` statements and thus produce unstructured decompiled output. Those statements are problematic because they make decompiled code harder to understand and less suitable for program analysis.**

In this paper, we present DREAM, the first decompiler to offer a `goto`-free output. DREAM uses a novel *pattern-independent* control-flow structuring algorithm that can recover all control constructs in binary programs and produce structured decompiled code without any `goto` statement. We also present

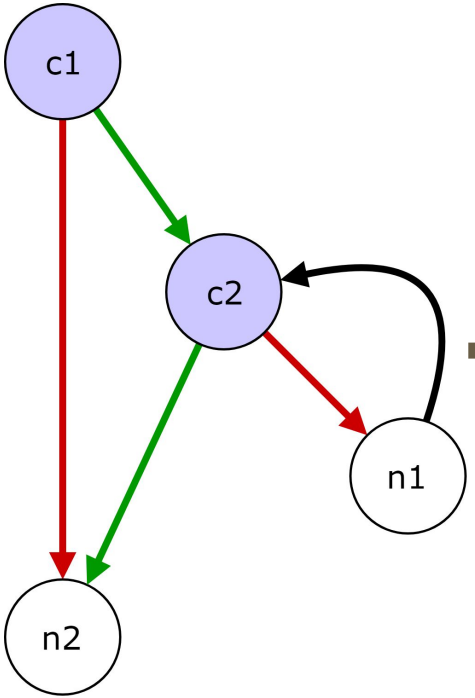
effective countermeasures and mitigation strategies requires a thorough understanding of functionality and actions performed by the malware. Although many automated malware analysis techniques have been developed, security analysts often have to resort to manual reverse engineering, which is difficult and time-consuming. Decompilers that can reliably generate high-level code are very important tools in the fight against malware: they speed up the reverse engineering process by enabling malware analysts to reason about the high-level form of code instead of its low-level assembly form.

Decompilation is not only beneficial for manual analysis, but also enables the application of a wealth of source-based security techniques in cases where only binary code is available. This includes techniques to discover bugs [5], apply taint tracking [10], or find vulnerabilities such as RICH [7], KINT [38], Chucky [42], Dowser [24], and the property graph approach [41]. These techniques benefit from the high-

Control Flow Structuring? (0)



Control Flow Structuring? (1)

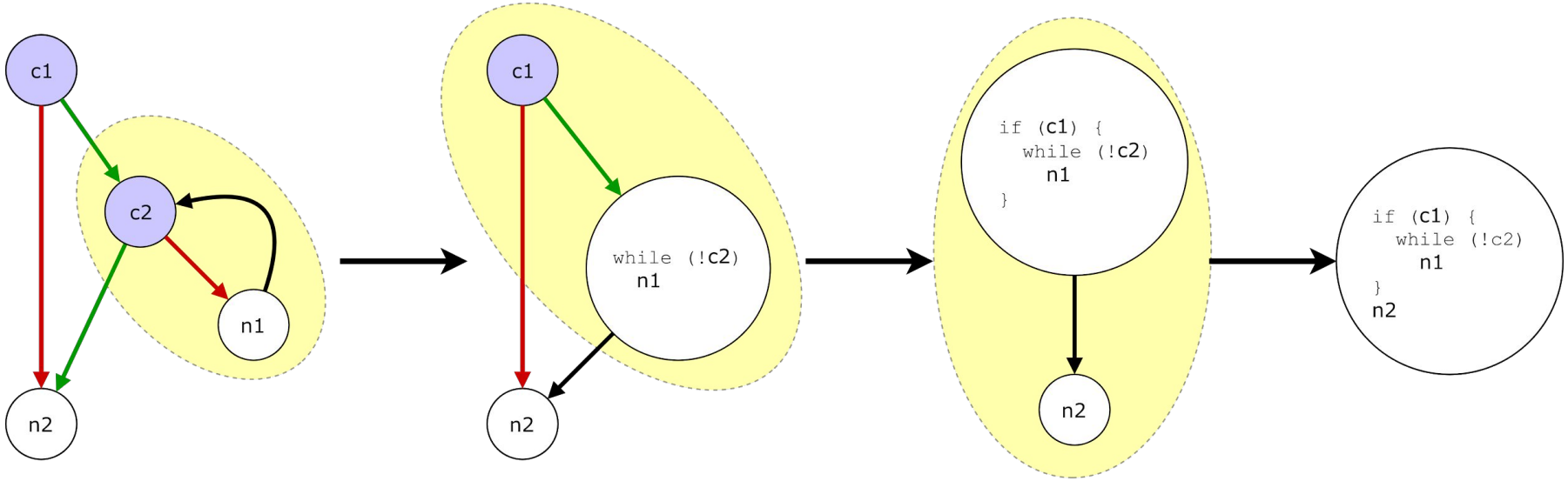



```
if (c1) {
   while (!c2)
      n1
}
n2
```

Structural Analysis

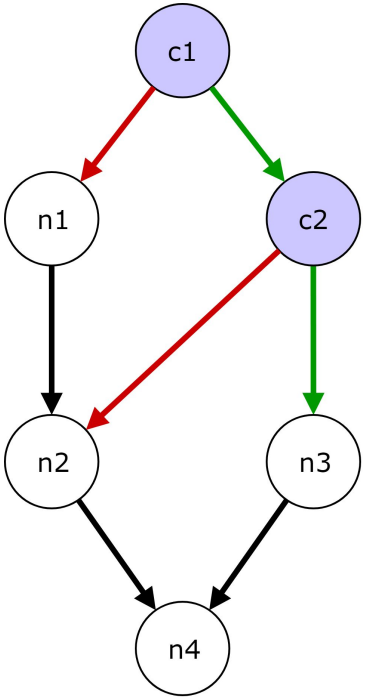
- Match subgraphs against a predefined set of patterns
- Used by Hex-Rays, RetDec, etc...

Structural Analysis Example



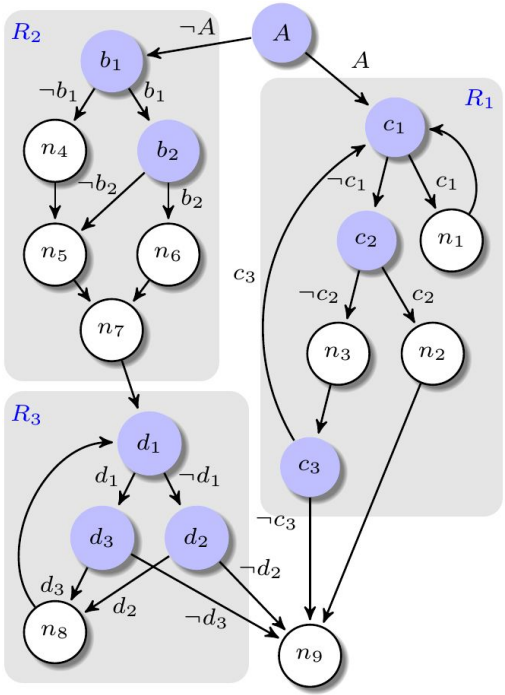
Problems with Structural Analysis

- Doesn’t work for arbitrary graphs
- Uses `goto` if no pattern is found

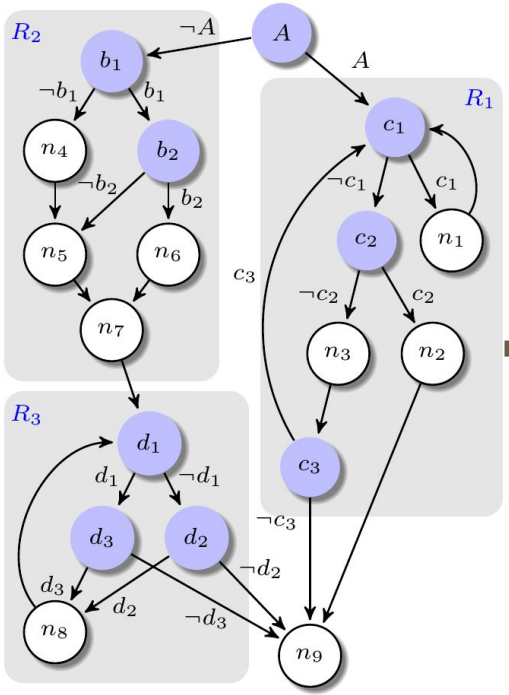


Pattern-Independent
Control Flow Structuring

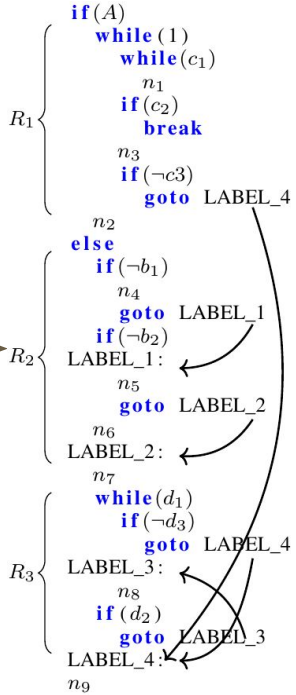
Example (0)


R_2
$\neg A$
A
b_1
$\neg b_1$
b_1
n_4
b_2
$\neg b_2$
b_2
n_5
n_6
n_7
A
R_1
c_1
$\neg c_1$
c_1
c_2
n_1
c_3
$\neg c_2$
c_2
n_3
n_2
c_3
R_3
d_1
d_1
$\neg d_1$
d_3
d_2
$\neg c_3$
$\neg d_2$
d_3
d_2
$\neg d_3$
n_8
n_9

Example (1)

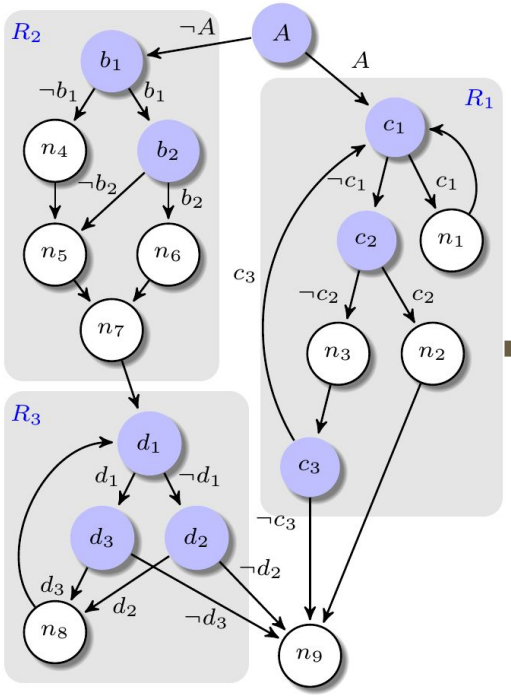


Hex-Rays

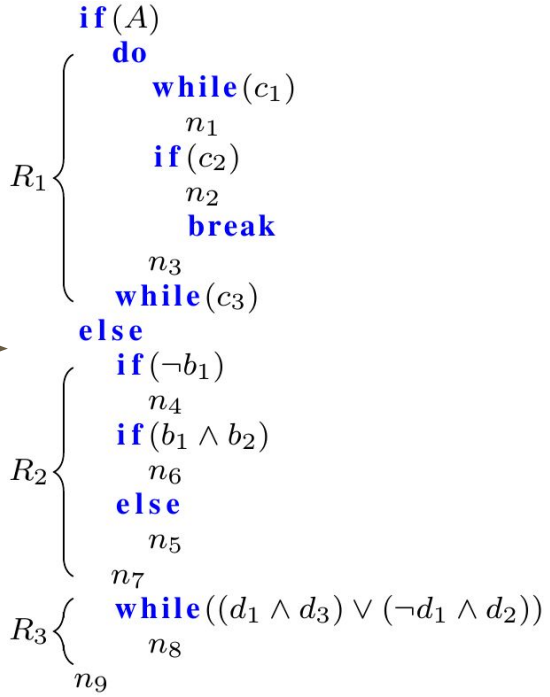


p4, *No More Gotos*

Example (2)



No More Gotos



p4, *No More Gotos*

LIVE DEMO!!!

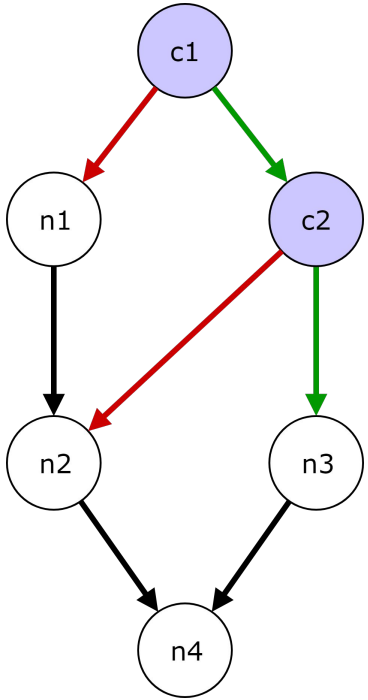
Algorithm Overview

1. For each node in post-order:
2. If this is the destination of a back-edge, structure this loop
3. If the set of nodes dominated by this node has a single successor, structure this region
4. Otherwise, continue iterating

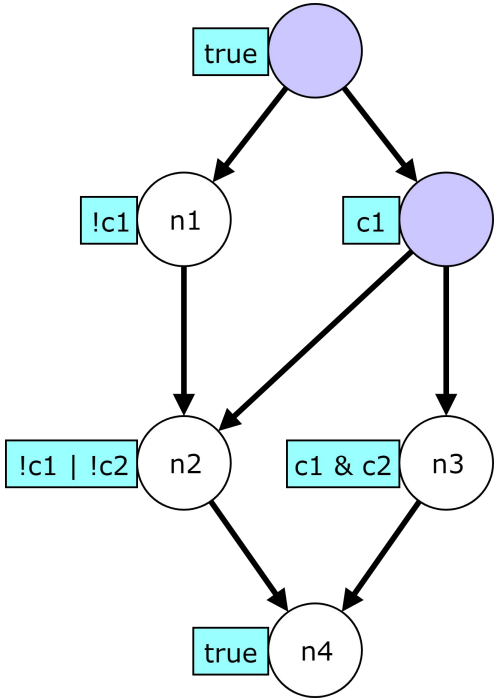
Structuring Acyclic Regions

1. Compute the *reaching condition* for each node
2. Move all nodes to a new graph and refine it
3. Make an AST node with all the nodes in topological order
4. Extract duplicated conditions

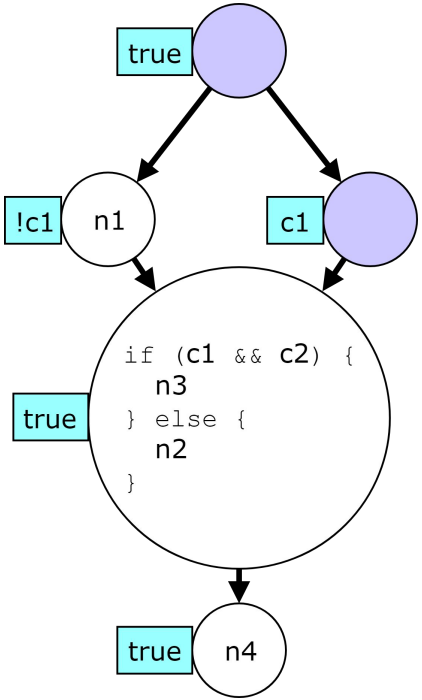
Structuring Acyclic Regions Example (0)



Structuring Acyclic Regions Example (1)



Structuring Acyclic Regions Example (2)



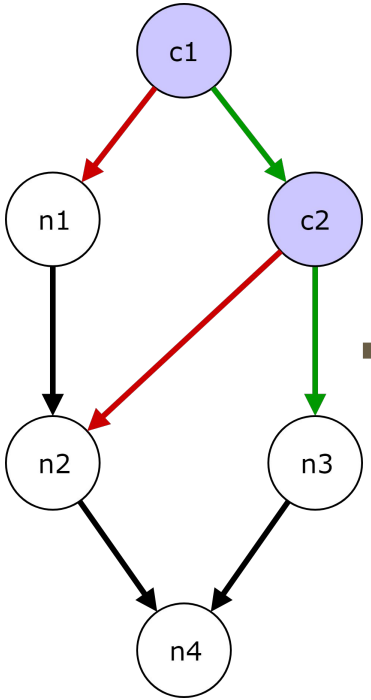
Structuring Acyclic Regions Example (3)

```
if (!c1) {
   n1
}
if (c1 && c2) {
   n3
} else {
   n2
}
n4
```

Structuring Acyclic Regions Example (4)

```
bool b = c1;
if (!b) {
  n1
}
if (b && c2) {
  n3
} else {
  n2
}
n4
```

Structuring Acyclic Regions Example (5)

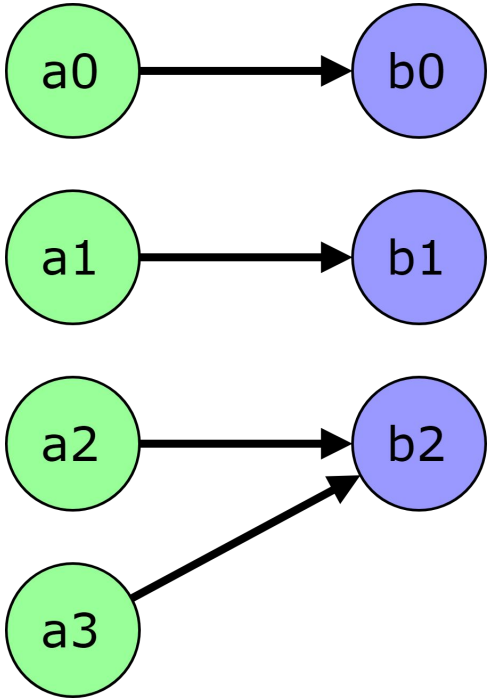



```
bool b = c1;
if (!b) {
   n1
}
if (b && c2) {
   n3
} else {
   n2
}
n4
```

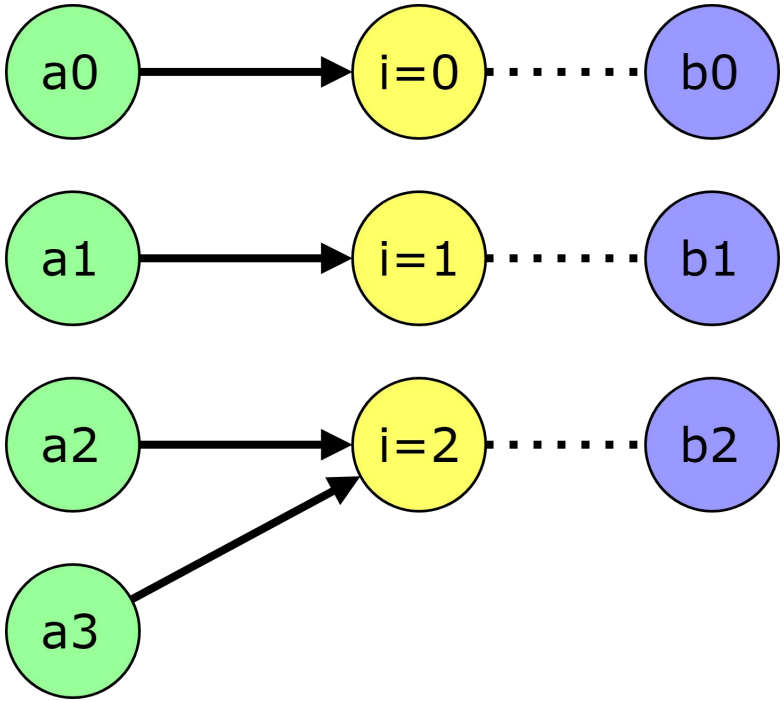
Structuring Cyclic Regions

1. Funnel abnormal entries
2. Add loop-dominated nodes into the loop itself
3. Replace all loop exits with `break`
4. Structure the loop body
5. Put the body in an infinite loop and refine the AST

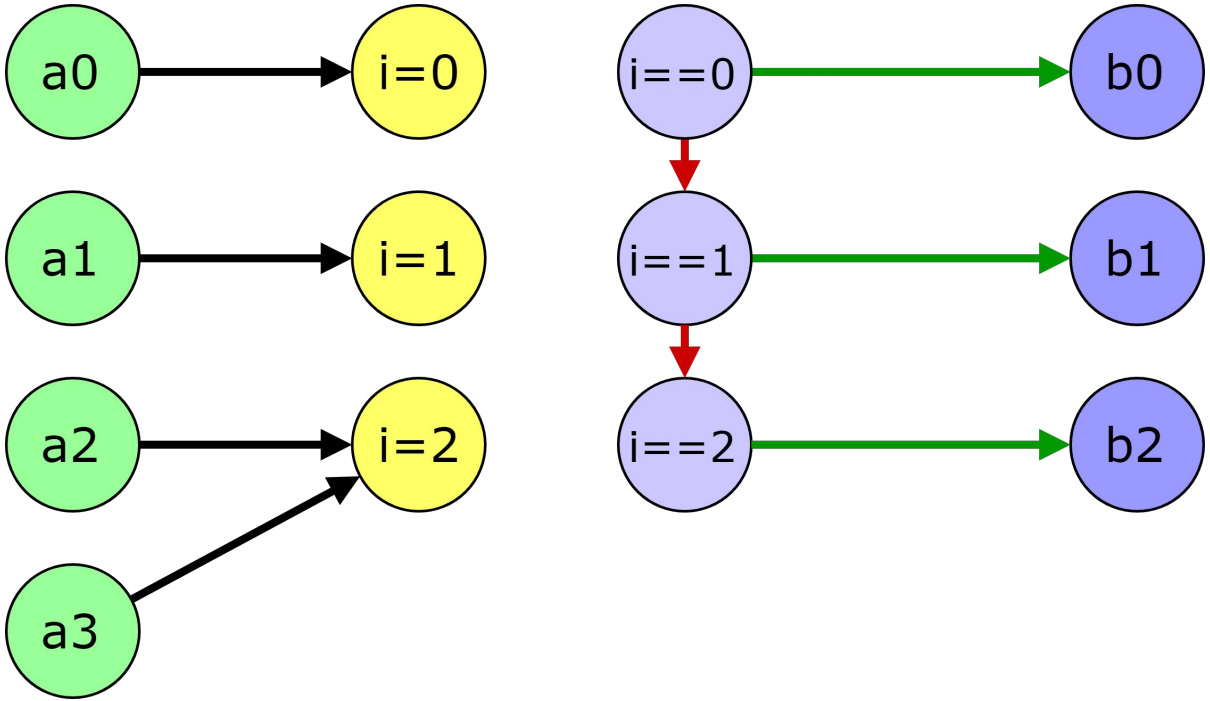
Funneling (0)



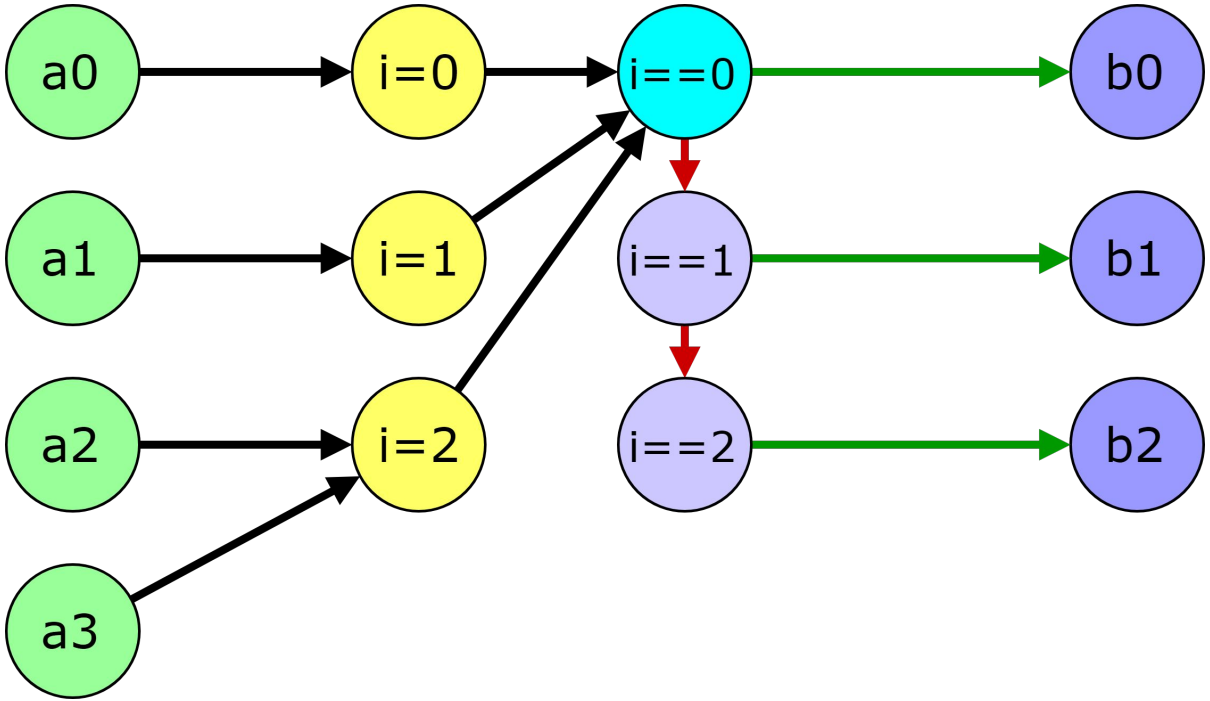
Funneling (1)



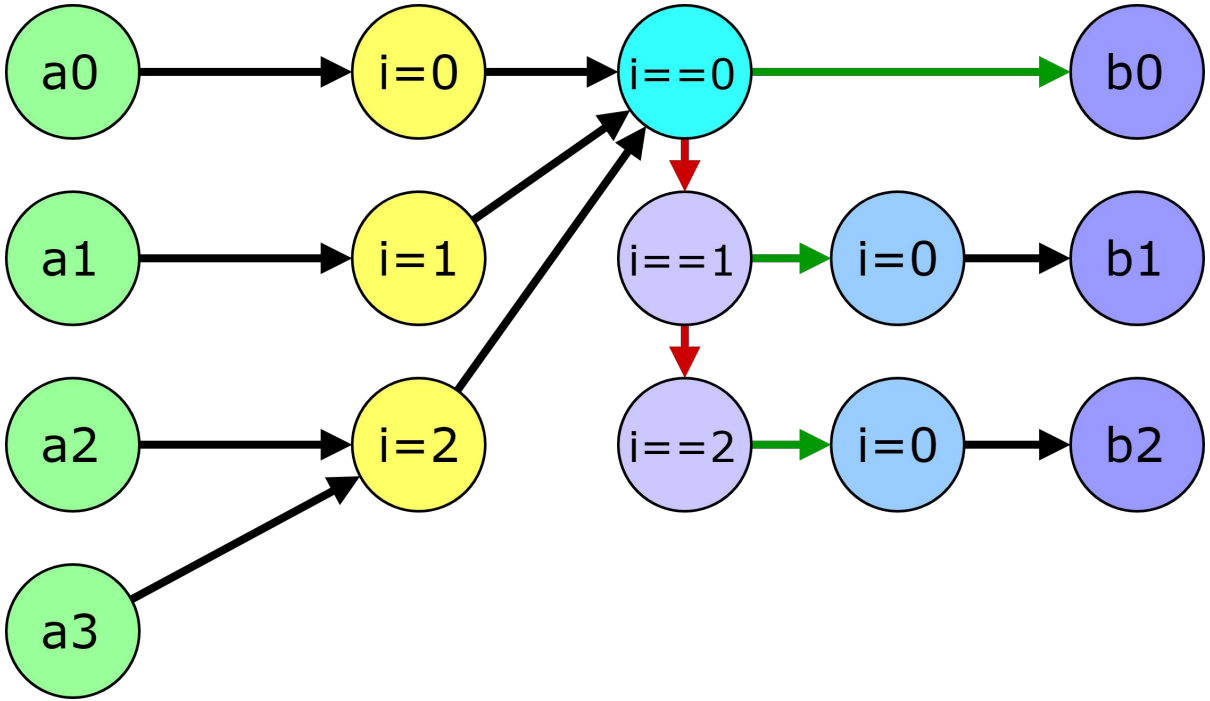
Funneling (2)



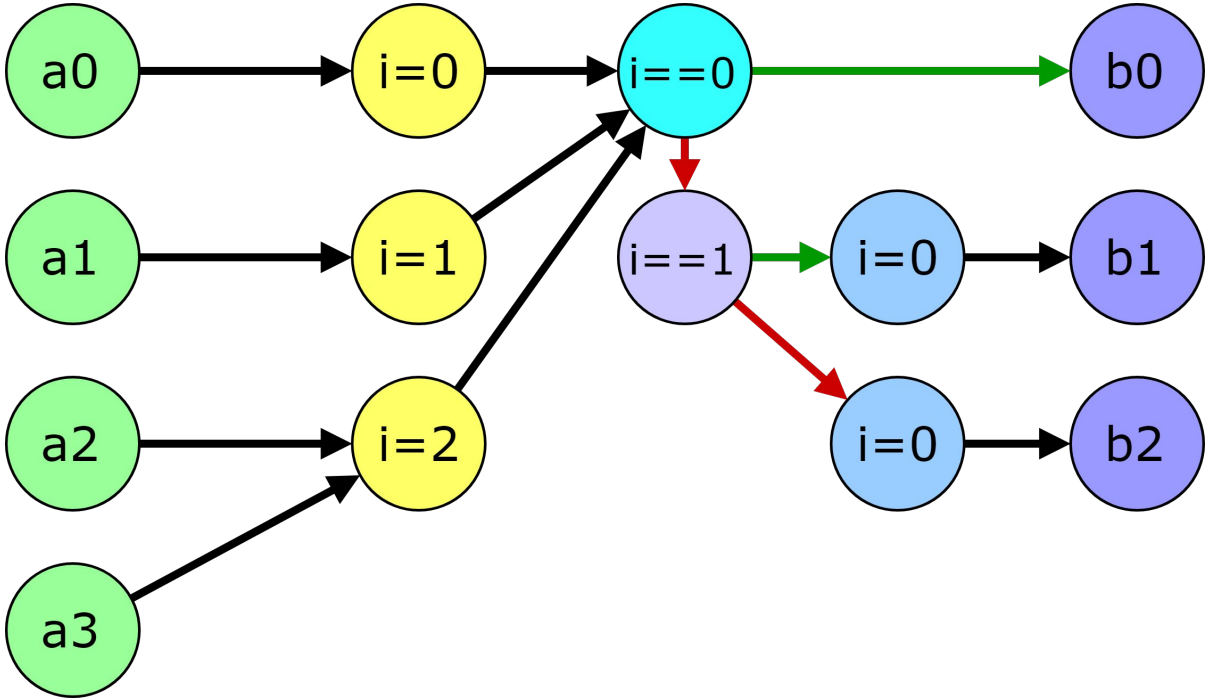
Funneling (3)



Funneling (4)



Funneling (5)



Loop AST Refinements

$$\frac{n_\ell = Loop\left[\tau_{\mathbf{endless}}, -, Seq\,[n_i]^{i\in 1..k}\right] \quad n_1 = \mathcal{B}_r^c}{n_\ell \rightsquigarrow Loop\left[\tau_{\mathbf{while}}, \neg c, Seq\,[n_i]^{i\in 2..k}\right]}\ \textsc{While} \qquad \frac{n_\ell = Loop\left[\tau_{\mathbf{endless}}, -, Seq\,[n_i]^{i\in 1..k}\right] \quad n_k = \mathcal{B}_r^c}{n_\ell \rightsquigarrow Loop\left[\tau_{\mathbf{dowhile}}, \neg c, Seq\,[n_i]^{i\in 1..k-1}\right]}\ \textsc{DoWhile}$$

$$\frac{n_\ell = Loop\left[\tau_{\mathbf{endless}}, -, Seq\,[n_i]^{i\in 1..k}\right] \quad \forall i \in 1..k-1 : \mathcal{B}_r \notin \sum[n_i] \quad n_k = Cond\,[c, n_t, -]}{n_\ell \rightsquigarrow Loop\left[\tau_{\mathbf{endless}}, -, Seq\left[Loop\left[\tau_{\mathbf{dowhile}}, \neg c, Seq\,[n_i]^{i\in 1..k-1}\right], n_t\right]\right]}\ \textsc{NestedDoWhile}$$

$$\frac{n_\ell = Loop\left[\tau_{\mathbf{endless}}, -, Seq\,[n_i]^{i\in 1..k}\right] \quad n_k = n'_k \Downarrow \mathcal{B}_r}{n_\ell \rightsquigarrow Seq\left[n_1, \ldots, n_{k-1}, n'_k\right]}\ \textsc{LoopToSeq}$$

$$\frac{n_\ell = Loop\left[\tau_{\mathbf{endless}}, -, Cond\,[c, n_t, n_f]\right] \quad \mathcal{B}_r \notin \sum[n_t] \quad \mathcal{B}_r \in \sum[n_f]}{n_\ell \rightsquigarrow Loop\left[\tau_{\mathbf{endless}}, -, Seq\left[Loop\,[\tau_{\mathbf{while}}, c, n_t], n_f\right]\right]}\ \textsc{CondToSeq}$$

$$\frac{n_\ell = Loop\left[\tau_{\mathbf{endless}}, -, Cond\,[c, n_t, n_f]\right] \quad \mathcal{B}_r \in \sum[n_t] \quad \mathcal{B}_r \notin \sum[n_f]}{n_\ell \rightsquigarrow Loop\left[\tau_{\mathbf{endless}}, -, Seq\left[Loop\,[\tau_{\mathbf{while}}, \neg c, n_f], n_t\right]\right]}\ \textsc{CondToSeqNeg}$$

Fig. 8: Loop structuring rules. The input to the rules is a loop node n_ℓ.

p9, *No More Gotos*

Loop AST Refinement Example (0)

```
while (1) {
  if (c1)
    n1
  else {
    if (c2) {
      n2
      break;
    }
    n3
    if (!c3)
      break;
  }
}
```

Loop AST Refinement Example (1)

```
while (1) {
  while (c1)
    n1
  if (c2) {
    n2
    break;
  }
  n3
  if (!c3)
    break;
}
```

Loop AST Refinement Example (2)

```
do {
  while (!c1)
    n1
  if (c2) {
    n2
    break;
  }
  n3
} while (c3);
```

Acknowledgements

- kriw
- sushant94
- chinmaydd
- XVilka

Thank You!

Questions?

References

- Blog post: https://radare.today/posts/gsoc_2018_radeco_cfs/
- *No More Gotos*: https://doi.org/10.14722/ndss.2015.23185