A Comb for Decompiled C Code

- We present a novel algorithm for control flow restructuring to restructure programs so that they can be emitted in C without resorting to `goto` statements.
- We implement the proposed approach on top of the `rev.ng` binary analysis framework.
- We compare the resulting decompiler with state-of-the arts decompilers.

Generality: Our solution should applicable on any arbitrary CFG.

Structured: The emitted output should not resort to employing unstructured programming statements, such as `gotos`.

Expressive: The decompiler should be able to emit a wide range of idiomatic C constructs, such as `while` and `do-while` loops, `switch` statements, `if-else` with short-circuited conditions.

The Control Flow Restructuring algorithm is designed in order to enforce the following properties:

- Two Successors.
- Two Predecessors.
- Loop Properties:
    - Single Entry.
    - Single Successor.
    - Single Retreating Edge.
    - Diamond Shape Property.

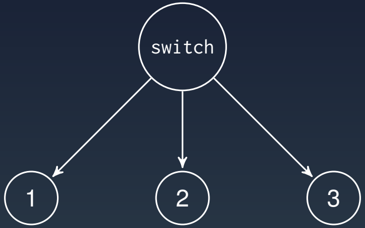
The Control Flow Restructuring algorithm is designed in order to enforce the following properties:

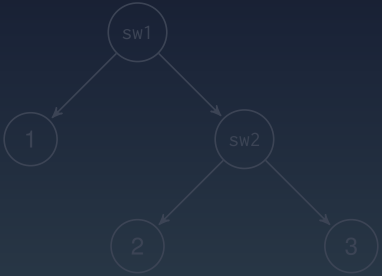
- Two Successors.
- Two Predecessors.
- Loop Properties:
    - Single Entry.
    - Single Successor.
    - Single Retreating Edge.
    - Diamond Shape Property.

The Control Flow Restructuring algorithm is designed in order to enforce the following properties:

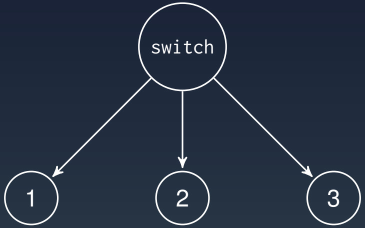
- Two Successors.
- Two Predecessors.
- Loop Properties:
  - Single Entry.
  - Single Successor.
  - Single Retreating Edge.
  - Diamond Shape Property.

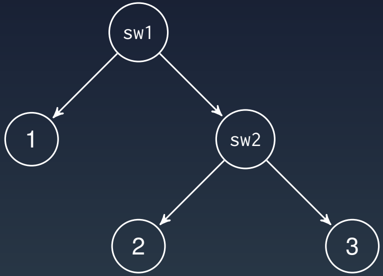
- Enforcing these properties is necessary, so that later stages in the decompilation pipeline, in particular the combing, can operate on the CFG.
- Specifically, our goal here is to transform switch statements in nested chains of `if-else if` statements.





- Enforcing these properties is necessary, so that later stages in the decompilation pipeline, in particular the combing, can operate on the CFG.
- Specifically, our goal here is to transform switch statements in nested chains of `if-else if` statements.

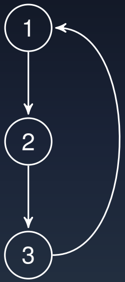




In order to be able to identify loops, and enforce the properties identified before, we need to introduce a new concept:

SCS (Strongly Connected Set). It is a region that corresponds to a cyclic region, and it is identified by a backedge.

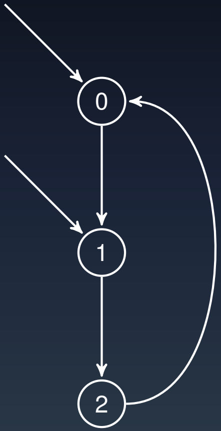
The identification of an SCS region is performed exploiting the concept of *backedge*. Specifically each backedge gives origin to a new SCS region, that is composed by all the nodes that are on paths built between the target node of the backedge and its source.

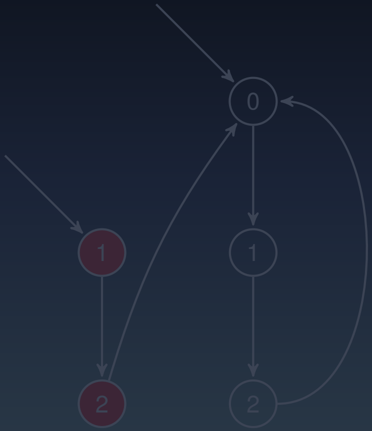


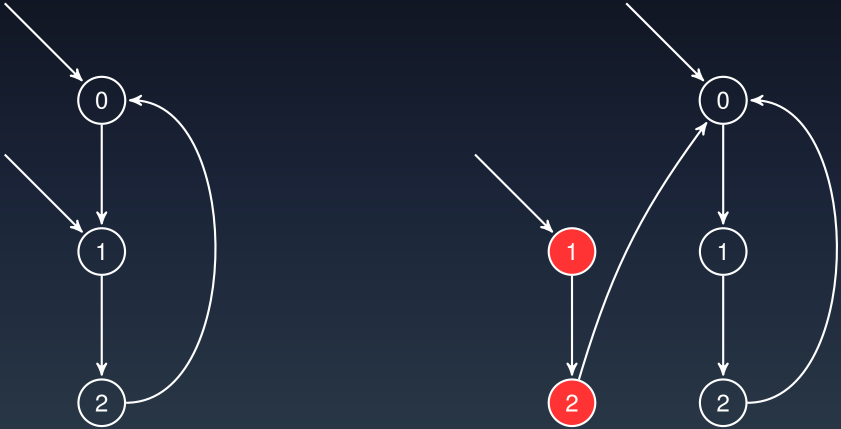
In general, we do not have the guarantee that a SCS presents a single entry node and a single successor node, and a single retreating edge. Specifically, we may have:

- **Abnormal entries** in the loop.
- **Abnormal exits** from the loop.
- **Abnormal retreating** edges.

For this reason, we designed some control-flow restructuring procedures to make any SCS tractable. We will see an example of how we remove the *abnormal entries*, but analogous transformations for dealing with the other two situations have been devised.

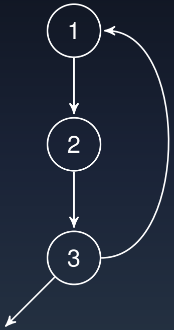

0
1
2

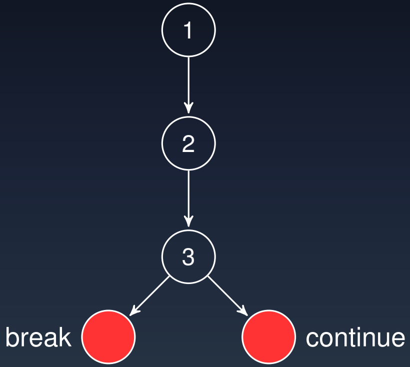

0
1
1
2
2


0
1
2
0
1
1
2
2

To make an SCS region acyclic, we do the following:

- All the nodes exiting the SCS are mapped on a `break` node.
- The retreating edges are mapped on a `continue` node.
- Exiting and retreating edges are disconnected.
- It is fundamental that we are able to virtually transform the cyclic regions in acyclic regions, so that the *combing* pass can operate on them.


1
2
3


1
2
3
break
continue

- At this point, we can introduce the *region collapse* process.
- We designed the *region collapse* process to reduce the complexity of the control-flow restructuring and AST reconstruction phases, and to transform the cyclic regions into acyclic ones.
- In fact, in this way, the later stages of the pipeline can operate on smaller (one cyclic region at a time), and (virtually) acyclic regions.

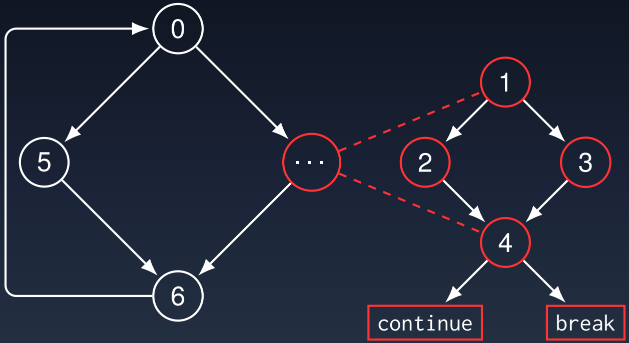


Figure: Collapsing nested DAG *Region*.

- *The comb* is a very important component of our decompilation pipeline.
- In particular, *the comb* is the component in charge of disentangling the control flow when it is not possible to recover further structure.
- Its task is to highlight the *diamond-shaped* regions.

- The *combing* underlying algorithm operates by using dominance and post-dominance analyses.
- Specifically, the main idea is that for each conditional node, we must enforce the property that, for each node between the conditional node itself and its immediate post-dominator, the conditional node must dominated all such nodes. If this is not true, we introduce clone nodes in order to enforce this property.
- After the *combing* has been enforced on the complete graph, we can move the the AST reconstruction phase, which as we will see will become trivial, thanks to the special form the CFG has assumed.

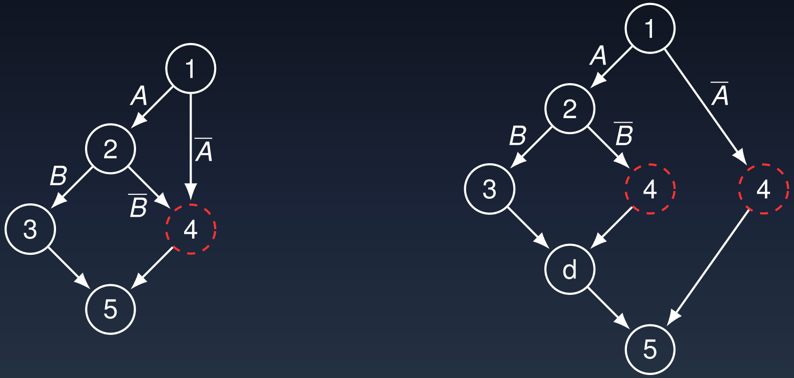


Figure: CFG Combing.

Our aim is to use as expressive as possible high level constructs in the decompiled code:

- Pre-compute a very simple and versatile AST: `if` and `while (1)`.
- Design post-processing passes on the AST to match complex statements: `while`, `do while`, `for`, `switch`.
- This approach makes this phase easily extendable, and highly modular.
- An example of this is the short-circuit match, which we will see later.

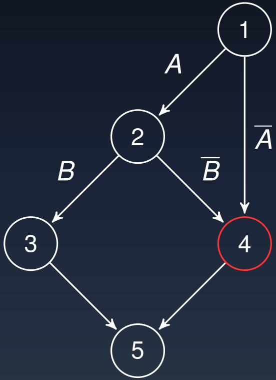
Once all the normalization phases have been applied to the CFG, we generate the corresponding AST. In the AST we have the following node types:

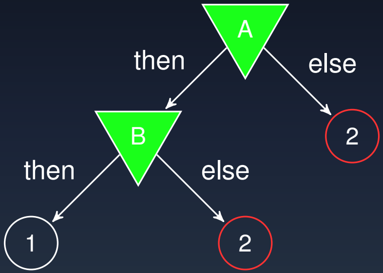
- Code node
- If node
- Loop node

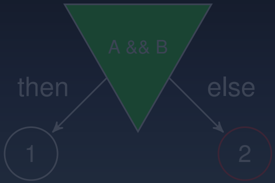
After the initial AST has been built, we implement further post-processing phases over the raw AST:

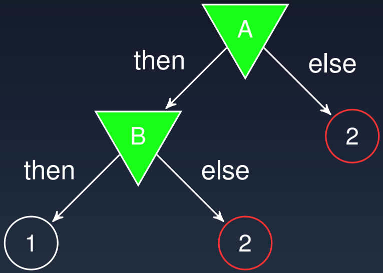
- The idea is to build a pipeline of passes each in charge of extracting high level control flow statements from the raw AST.
- In this way, the reconstruction of the statements is a iterative refinement of the AST which is enriched at every step.
- Therefore, the introduction of a new potential post-processing pass becomes easy and does not overturn the whole decompiler structure.

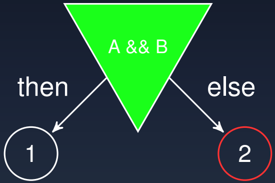
We designed a post-processing phase whose purpose is to reconstruct the *short-circuited* `if` statements, using the information about duplication generated during the comb phase. Specifically, we inspect nested `if` AST nodes that present in their branches code nodes that generated from the duplication of an original single node.


1
A
$\overline{A}$
2
B
$\overline{B}$
3
4
5


A
then
else
B
2
then
else
1
2


A && B
then
else
1
2


A
then
else
B
2
then
else
1
2


A && B
then
else
1
2

- Our result data collection phase operates in two ways:
- As first thing, we collect the metric representing the code **duplication** introduced by our solution. This is done internally in our decompilation pipeline.
- On the other hand, to evaluate the quality of the code generated by the decompilers, we analyze the **pseudocode** generated as output.
- In particular, for all the three decompilers, we compute the **cyclomatic complexity** of the code after re-parsing it with the LLVM frontend.
- The evaluation has been conducted using the **GNU Coreutils** as benchmark suite.

	-O0	-O1	-O2	-O3
All	1.07×	1.10×	1.15×	1.32×
No-Goto	1.04×	1.08×	1.12×	1.25×

Table: Size increment metrics (over the original size) for the functions over different optimization levels.

	-O0			-O1			-O2			-O3		
	revng-c	IDA	Ghidra	revng-c	IDA	Ghidra	revng-c	IDA	Ghidra	revng-c	IDA	Ghidra
Cyclomatic Complexity	+11%	+12%	+16%	+13%	+17%	+20%	+36%	+60%	+72%	+78%	+86%	+94%
Gotos	0	1010	1370	0	2370	2622	0	2082	2062	0	2119	2282
Matched functions	93%			91%			89%			81%		

Table: Comparison between `revng-c`, IDA, and Ghidra.

If you have any question, you can write an e-mail to
`andreagus at rev.ng`