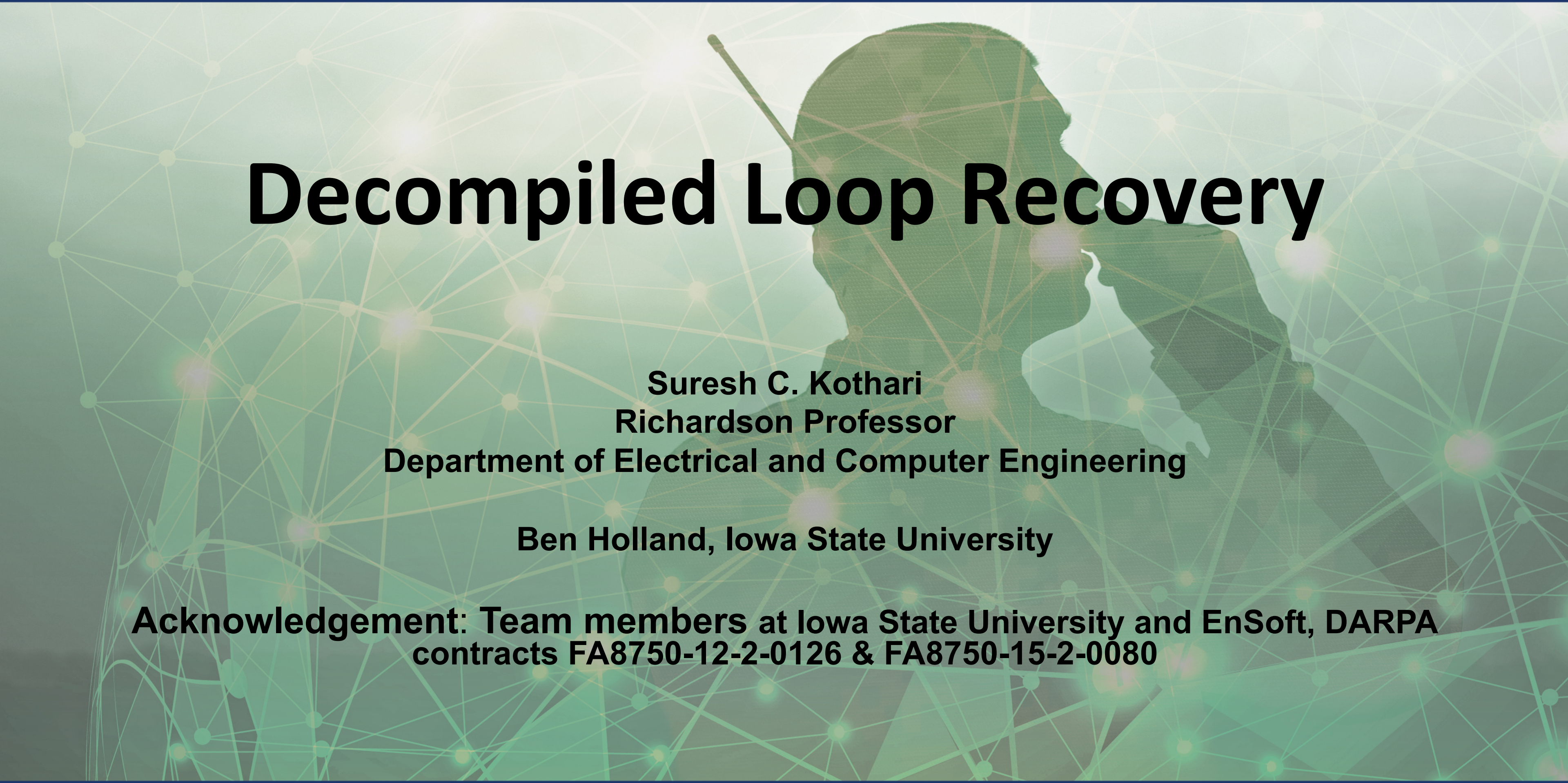
MILCOM 2016

SECURE COMMUNICATIONS AT THE SPEED OF CYBER



Decompiled Loop Recovery

Suresh C. Kothari
Richardson Professor
Department of Electrical and Computer Engineering

Ben Holland, Iowa State University

Acknowledgement: Team members at Iowa State University and EnSoft, DARPA contracts FA8750-12-2-0126 & FA8750-15-2-0080

BALTIMORE, MD • NOVEMBER 1–3, 2016

Motivation for Recovering Loops

- Most of the execution time is spent in loops
- 90/10 law
  - 90% of the time is spent in 10% of the code
  - 10% of the time is spent in the remaining 90% of the code

Challenge

- In compiled code high-level source constructs such as *for loops* and *while loops* do not exist.
  - Low-level code consists of goto's and labels

Goal: Identify loops in Control Flow Graphs (CFGs)

Loop Definition

A loop in the CFG:

- Has a set of child nodes (basic blocks)
- A loop has a *loop header* such that
  - Control to all child nodes in the loop always goes through the loop header
  - Has a back edge from one of its child nodes to the loop header

Remember

- Node X *dominates* node Y if all paths from the entry node to Y go through X
- A *depth-first search* of a graph starts at the root (CFG entry node) and explores as far as possible along each branch before backtracking.

Loop Recovery Intuitions

- Header of a loop dominates all child nodes in loop body
- Back edges are edges whose heads dominate their tails
  - An edge X→Y such that Y dominates X
- Loop identification is essentially back edge identification

Loop Recovery Algorithm

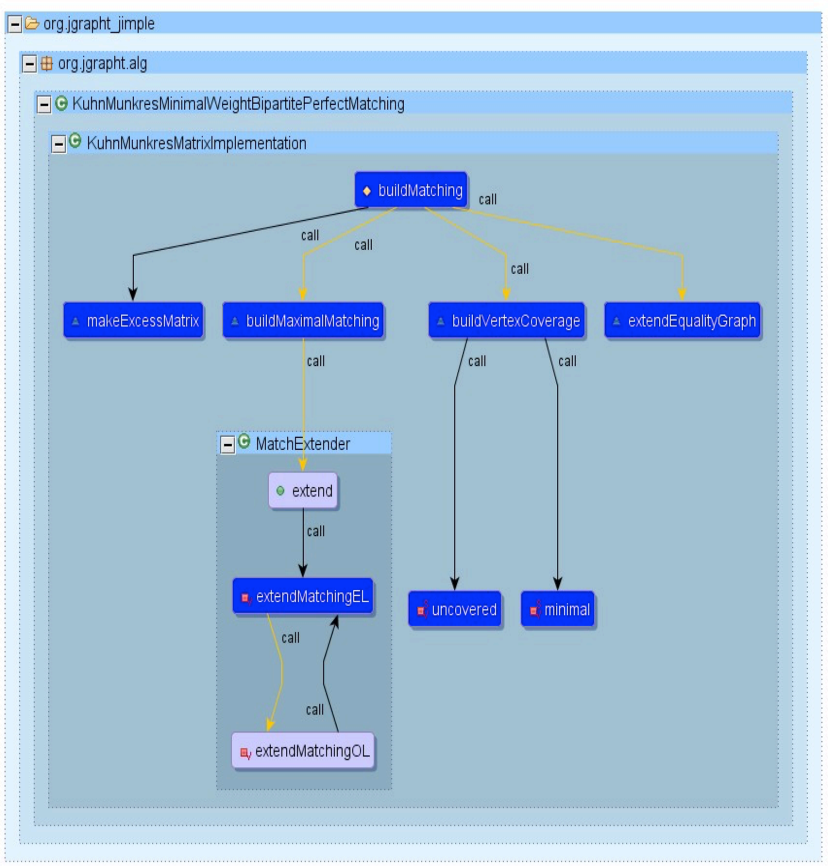
```
foreach node H in dominator tree
    foreach node N such that ∃ an edge N→H
        define loop:
            header = H
            back edge = N→H
            loop body = nodes found in a backwards
                             DFS traversal from N to H
```

Loop Recovery Algorithm

- DLI algorithm described in [1] presents an efficient algorithm for identifying loops in irreducible graphs.

[1] Wei, Tao, et al. "A new algorithm for identifying loops in decompilation." International Static Analysis Symposium. Springer Berlin Heidelberg, 2007.

Loop Call Graph



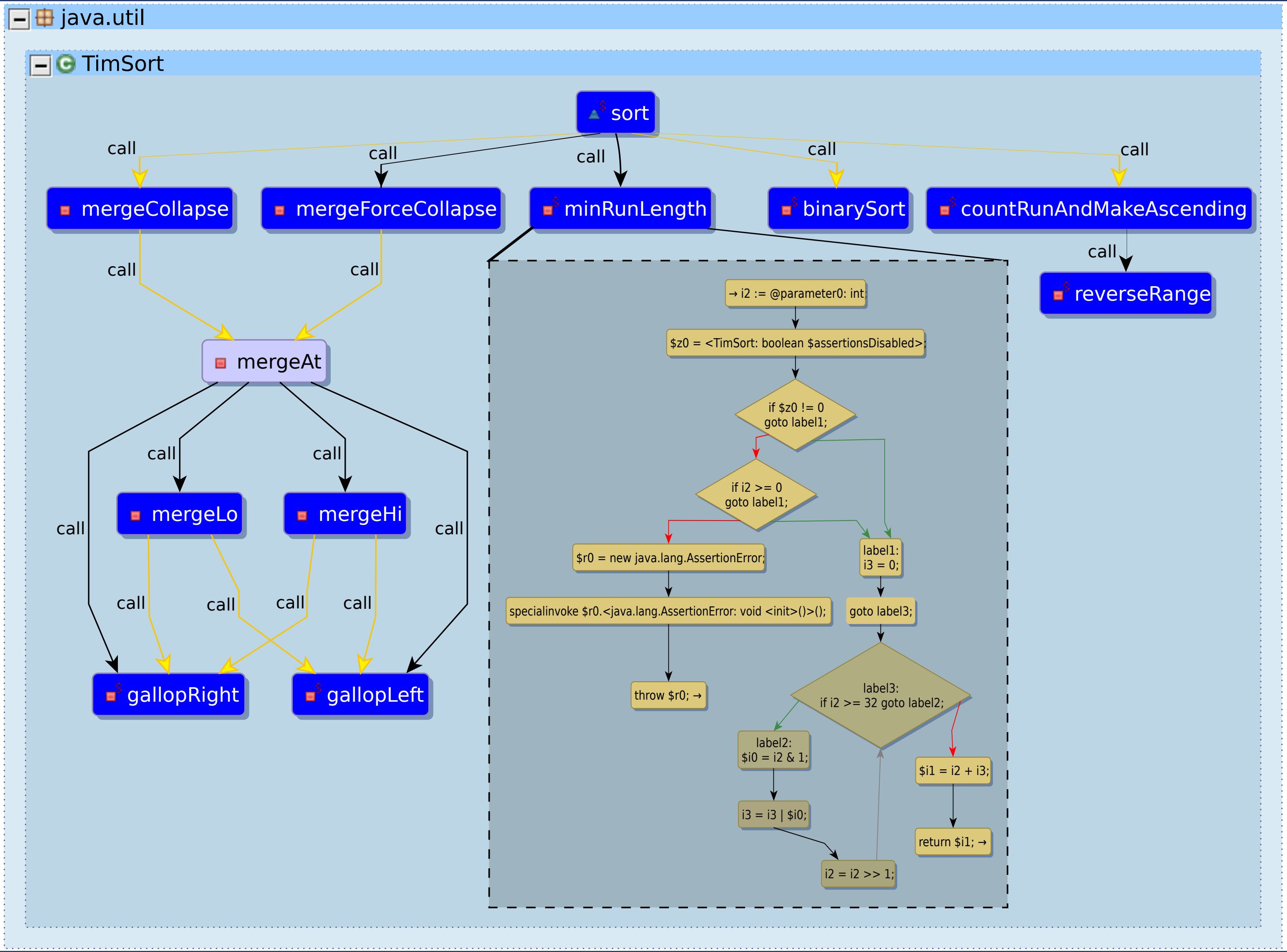
Called Inside Loop

Called Outside Loop

Nodes:
- Methods containing loops (blue)
- Methods reaching methods containing loops (white)

Edges:
- Call relationships
- Color attributes to show placement of call site in loop

Loop Call Graph



```
→ i2 := @parameter0: int
$z0 = <TimSort: boolean $assertionsDisabled>
if $z0 != 0 goto label1;
if i2 >= 0 goto label1;
$r0 = new java.lang.AssertionError;
label1:
i3 = 0;
specialinvoke $r0.<java.lang.AssertionError: void <init>()>();
goto label3;
throw $r0; →
label3:
if i2 >= 32 goto label2;
label2:
$i0 = i2 & 1;
$i1 = i2 + i3;
i3 = i3 | $i0;
return $i1; →
i2 = i2 >> 1;
```