AutoHacking with Phoenix Enabled Data Flow Analysis

Richard Johnson | switech@microsoft.com

Topics

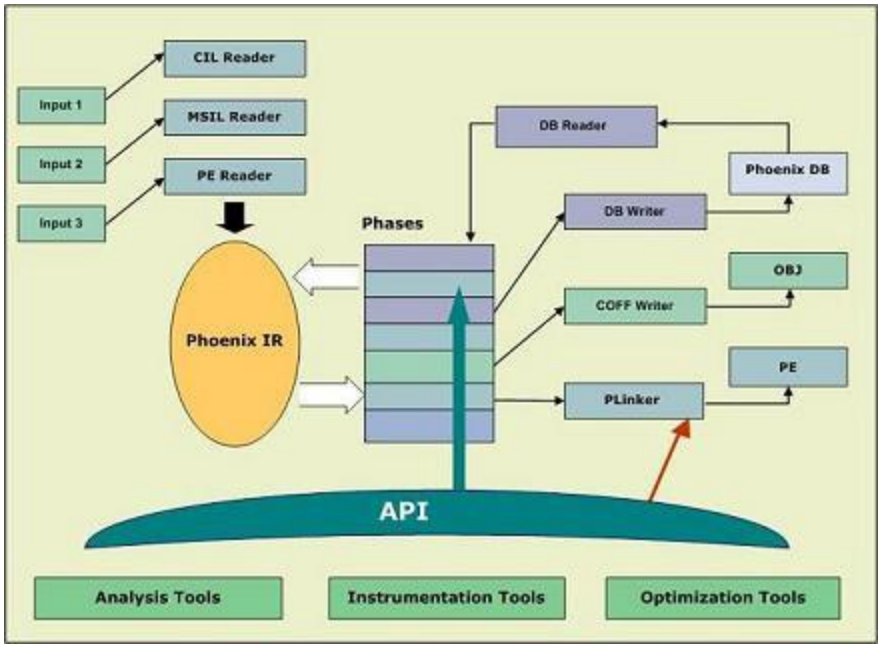
- Phoenix
  - Architecture
  - Fundamentals
- Program Analysis
  - Call Flow
  - Control Flow
  - Data Flow
- Applied Program Analysis
  - API Path Validation
  - Integer Overflow Detection
  - Syntax Model Inference

Introducing Phoenix

- Framework for building compilers and program analysis tools
- Collaborative project between the Microsoft Research, Visual C++, and .NET Common Language Runtime groups at Microsoft
- Foundation for the next generation of Microsoft development tools

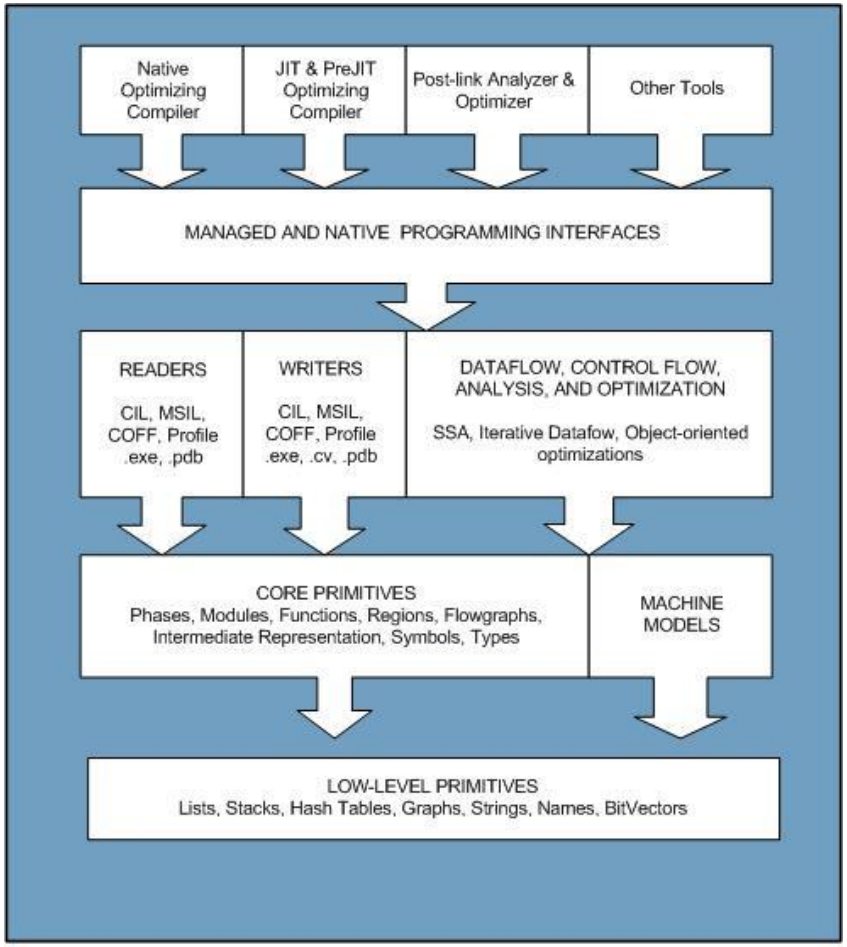
Phoenix Architecture

- Inputs are converted to an intermediate representation (IR)
- Phoenix API allows compiler plug-ins or standalone tools to add or hook phases of IR creation

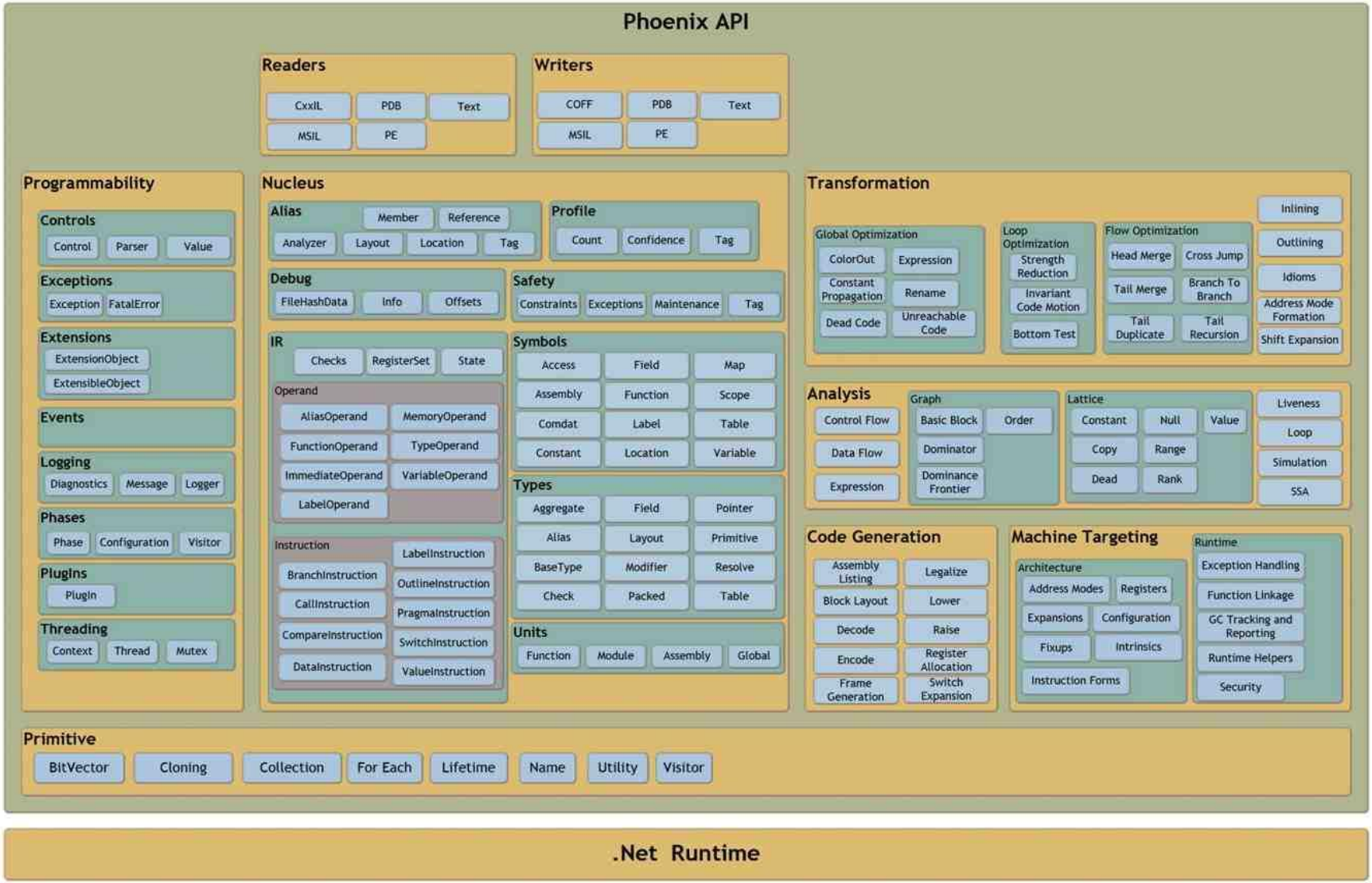


Phoenix Architecture

- Phases process IR to provide abstractions such as call graphs, flow graphs, region graphs, single static assignment (SSA) annotations



Phoenix Architecture



Phoenix Applications

- Compiler development
  - Optimization
  - Retargeting
- Binary Instrumentation
  - Profiling/Code coverage
  - Binary protection/obfuscation
- Program Analysis
  - Model inference
  - Vulnerability detection

Using Phoenix

- Load targets manually or via plug-ins
- Use phase lists or raise binaries manually

```
PEModuleUnit module = null;

module = PEModuleUnit.Open(path);
module.LoadPdb();
module.LoadGlobalSymbols();

Phx.Symbols.Table table = module.SymbolTable;
Phx.Symbols.LocalIdMap localMap = table.LocalIdMap;
Phx.Collections.IdToSymbolMap symbolMap = localMap.InternalMap;
Phx.Collections.IdToSymbolMap.Iterator iterator =
           new Phx.Collections.IdToSymbolMap.Iterator(symbolMap);

while (iterator.MoveNext())
{
  if (iterator.CurrentValue.IsFunctionSymbol)
  {
    FunctionSymbol function =  iterator.CurrentValue.AsFunctionSymbol;
```

Using Phoenix

- Load targets manually or via plug-ins
- Use phase lists or raise binaries manually

```
public class PlugIn : Phx.PlugIn {

public override void BuildPhases
(Phx.Phases.PhaseConfiguration config)
{
  Phx.Phases.Phase funcNamesPhase;
  funcNamesPhase = Phase.New(config);

  Phx.Phases.Phase encodingPhase =
     config.PhaseList.FindByName("Encoding");

  encodingPhase.InsertBefore(funcNamesPhase);
}

public override System.String NameString
{
   get { return "FuncNames"; }
}
```

```
public class Phase : Phx.Phases.Phase {

public static Phx.Phases.Phase
New (Phx.Phases.PhaseConfiguration config)
{
  Phase phase = new Phase();
  phase.Initialize(config, "FuncNames");
  return phase;
}

protected override void
Execute (Phx.Unit unit)
{
  if (!unit.IsFunctionUnit) return;

   Phx.FunctionUnit function =
           unit.AsFunctionUnit;
```

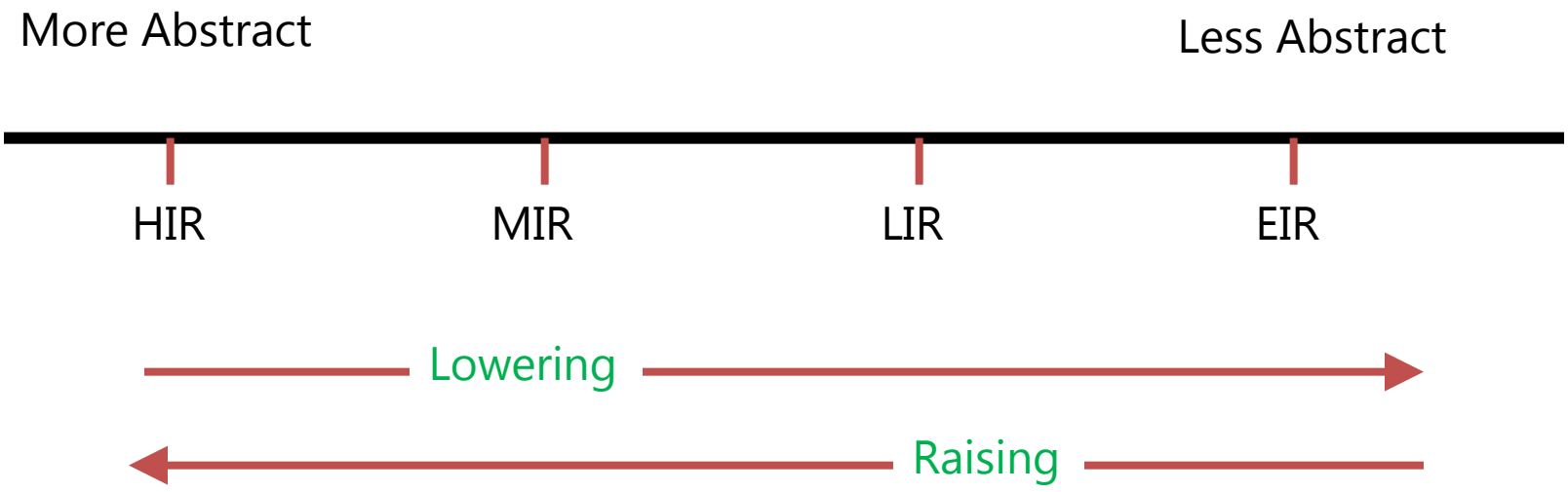
Using Phoenix

- Units
  - Programs, Assemblies, Modules, Functions
- Types
  - Primitives, Symbolic
- Symbols
  - Static, Dynamic

- Intermediate Representation
  - Primary abstraction of program semantics
  - Composed of Instructions and Operands
  - Three distinct levels of abstraction

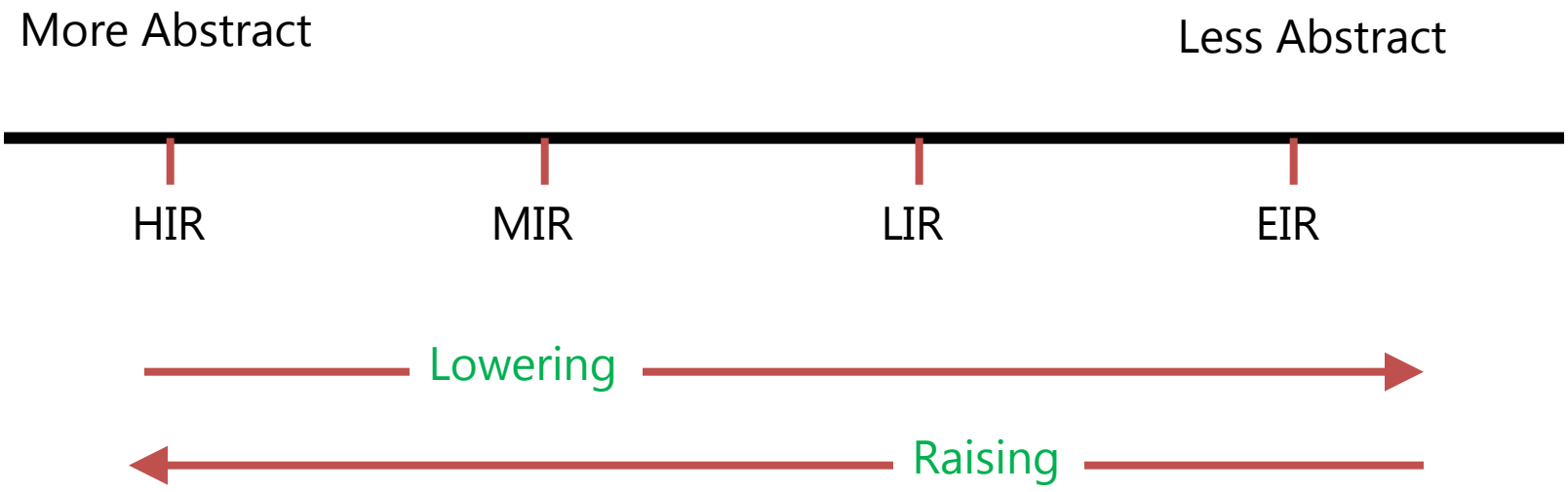
Phoenix Intermediate Representation

- High-level IR (HIR)
  - Architecture Independent
  - Abstract instructions represent runtime indirection
  - Operands refer to logical resources



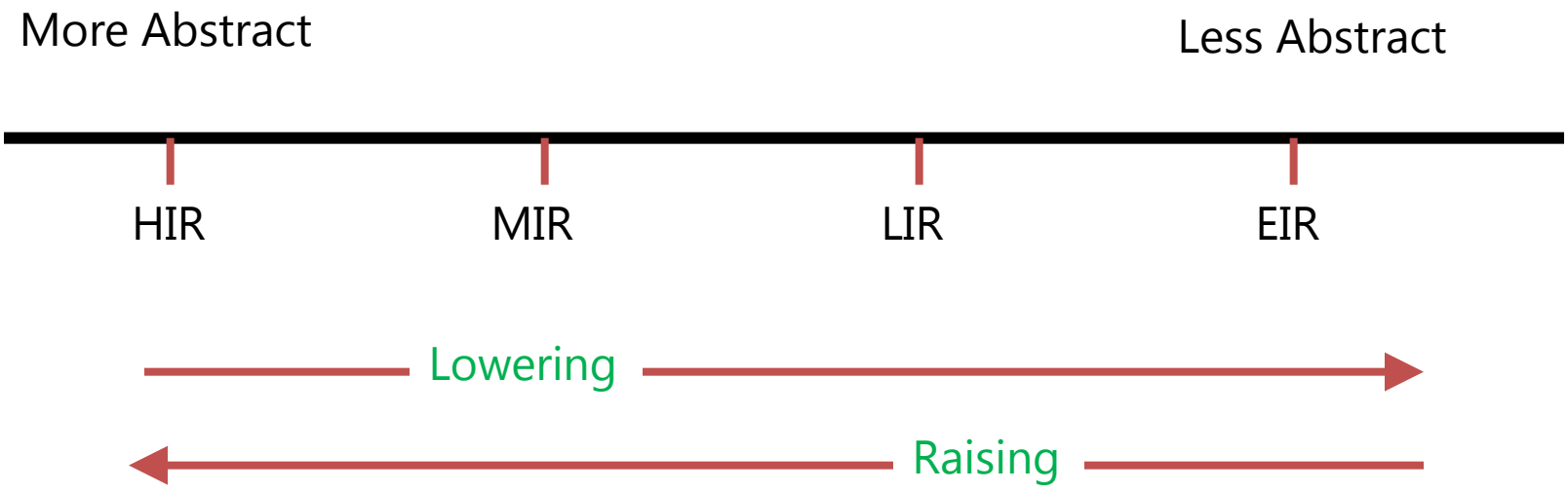
Phoenix Intermediate Representation

- Mid-level IR (MIR)
  - Architecture Independent
  - Runtime logic explicitly defined
  - Operands still refer to logical resources



Phoenix Intermediate Representation

- Low-level IR (LIR)
  - Architecture dependent
  - Control flow explicit
  - Operands refer to logical or physical resources



Phoenix Instructions

- Code or Data object
- Source and Destination Operands
- Annotation Operands

- Types
  - Label
  - Value
  - Compare
  - Branch
  - Call

High-level IR

```
$L1: (references=0)
   {*StaticTag}, {*NotAliasedTag} = START _main(T)
_main: (references=1)
   _argc, _argv    = ENTERFUNCTION
   t273            = COMPARE(GT) _argc, 1
                   CONDITIONALBRANCH(True) t273, $L7, $L6
$L7: (references=1)
   _message        = ASSIGN &$SG3745
                     GOTO $L8
```

Low-level IR

```
$L1: (references=0)
   {*StaticTag}, {*NotAliasedTag} = START WriteData(T)
WriteData: (references=1)
                       ENTERFUNCTION
   Local0, {ESP}     = push EBP
   EBP               = mov ESP
   tv144-            = mov 4112(0x00001010)
   {ESP}             = call _chkstk, {ESP}

   offset            = mov 8
   tv144-, {ESP}     = call CreateHeader, {ESP}

   header            = mov tv144-
   $Stack+32928, {ESP} = push 8
   $Stack+32960, {ESP} = push header
   tv144-            = lea &buf*
```

Phoenix Operands

- Instruction arguments
- Temporary Variables
- Alias Tags
- Alias Operands

- Types
  - Memory
  - Constants
  - Variables
  - Functions
  - Labels
  - Alias Sets

High-level IR

```
$L1: (references=0)
   {*StaticTag}, {*NotAliasedTag} = START _main(T)
_main: (references=1)
   _argc, _argv    = ENTERFUNCTION
   t273            = COMPARE(GT) _argc, 1
                   CONDITIONALBRANCH(True) t273, $L7, $L6
$L7: (references=1)
   _message        = ASSIGN &$SG3745
                     GOTO $L8
```

Low-level IR

```
$L1: (references=0)
   {*StaticTag}, {*NotAliasedTag} = START WriteData(T)
WriteData: (references=1)
                       ENTERFUNCTION
   Local0, {ESP}     = push EBP
   EBP               = mov ESP
   tv144-            = mov 4112(0x00001010)
   {ESP}             = call _chkstk, {ESP}

   offset            = mov 8
   tv144-, {ESP}     = call CreateHeader, {ESP}

   header            = mov tv144-
   $Stack+32928, {ESP} = push 8
   $Stack+32960, {ESP} = push header
   tv144-            = lea &buf*
```

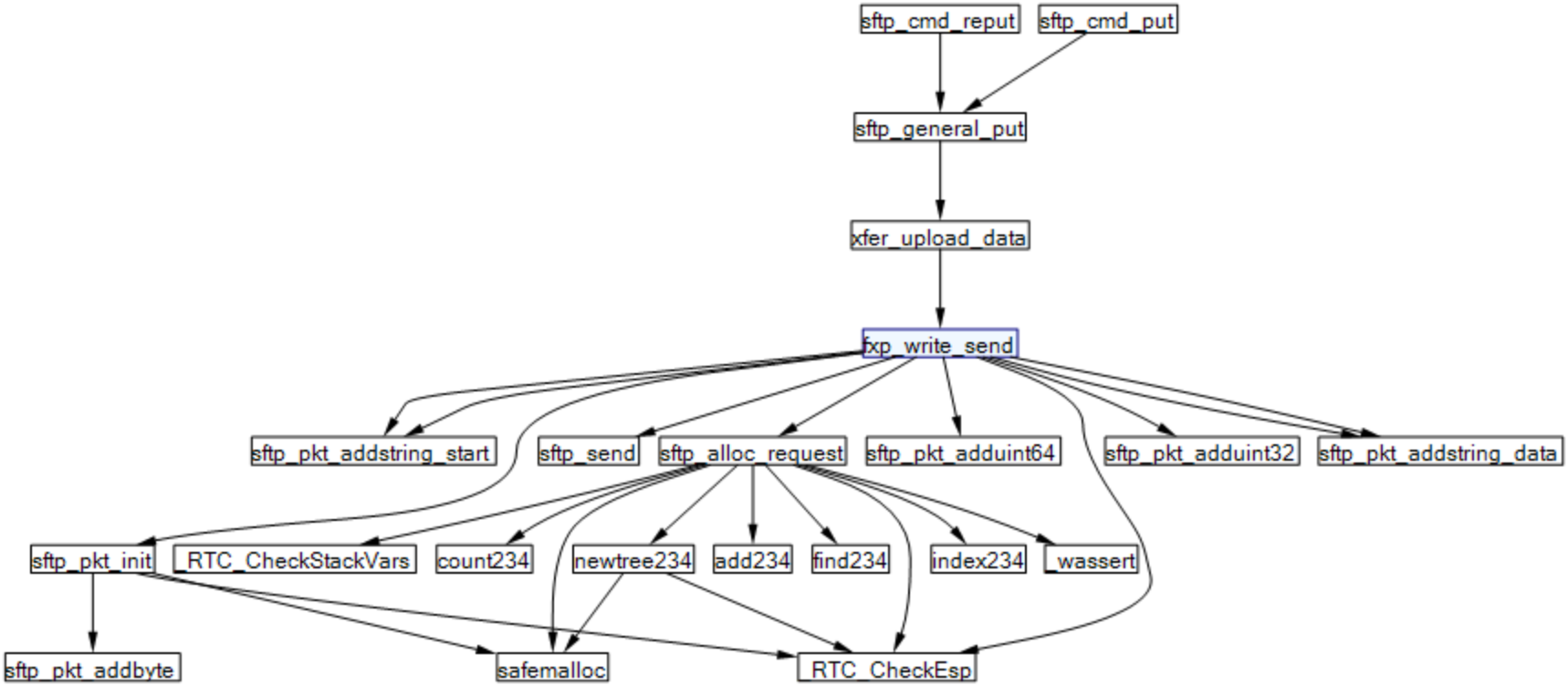
Phoenix Alias Package

- Alias System provides a memory model for static program analysis
- Aliases abstract memory and register use by assigning tags to discrete locations
- Alias Operands added to represent implicit effects of an instruction on memory

```
[ESP], {ESP} = push _message[EBP]
```

Call Graphs

- A Call Graph is a visual representation of call relationships between functions



Call Graphs

- Traditional call graph generation

Collect all call edges

```
foreach(Function in ModuleFunctions)
   foreach(CallInstruction in Instructions)
      AddCallEdge(Function, CallTarget))
```

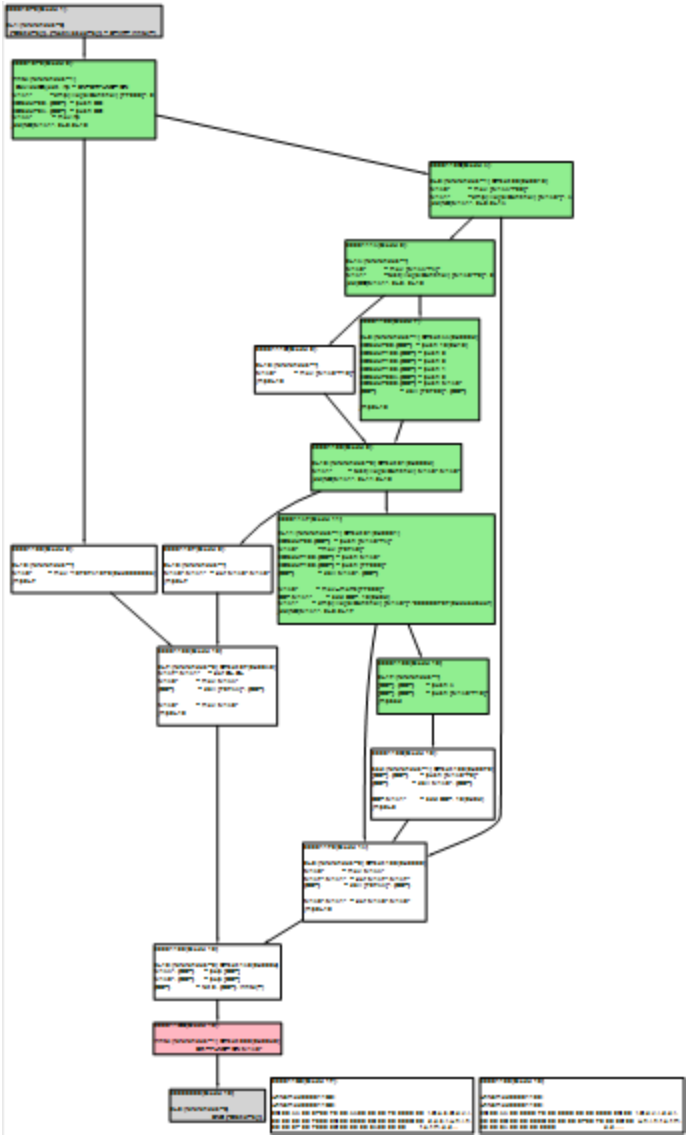
Find edges for Function

```
foreach(Edge in CallEdges)
   if(Target == Function)
      EdgesTo.Add(Edge)
   if(Source == Function)
      EdgesFrom.Add(Edge)
```

- Phoenix includes a Call Graph Package that provides module or program level function relationships

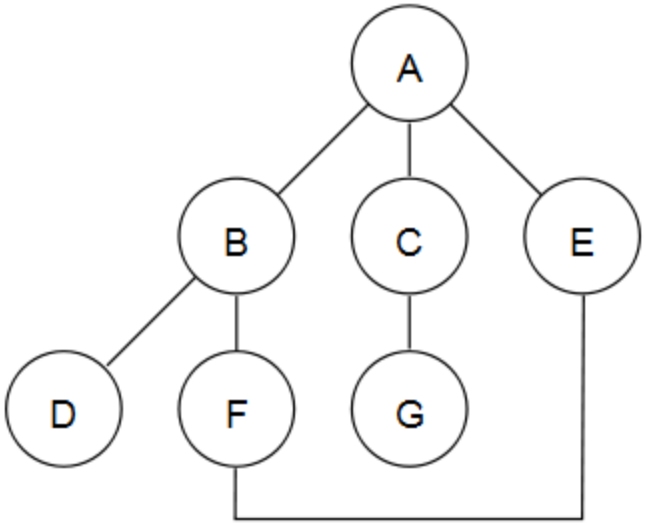
Control Flow Graphs

- Control Flow Graph are visual representations of branch relationships between basic blocks
- Phoenix provides a Control Flow Graph package that specifies edge types, node types, node dominance



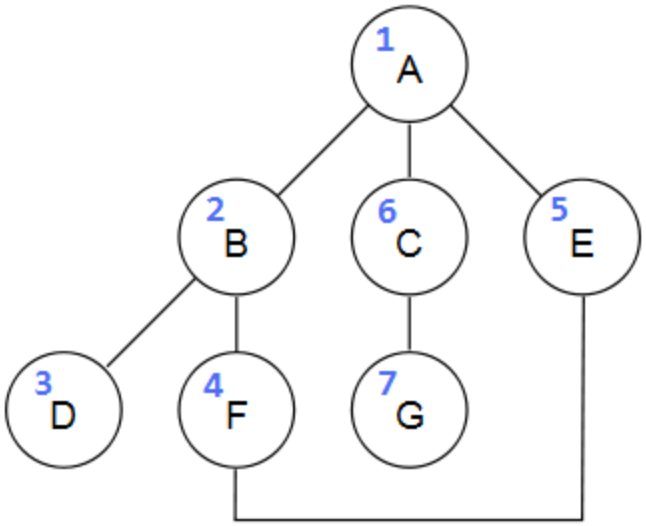
Graph Traversal

- Depth First Search
  - Visit nodes following edges as deep as possible before returning to the next edge



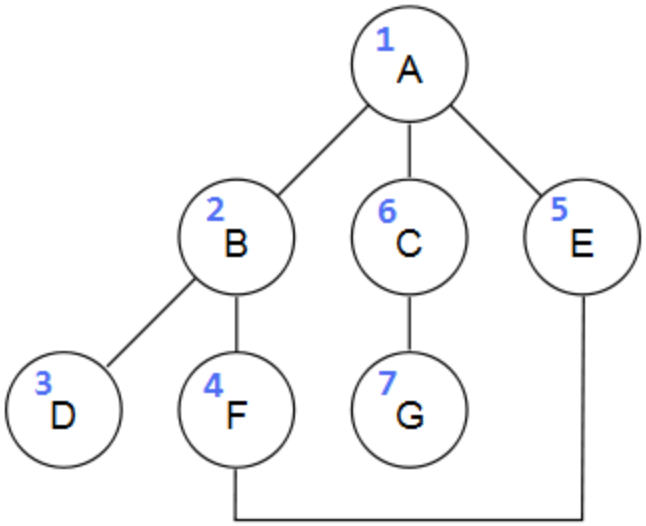
Graph Traversal

- Depth First Search
  - Visit nodes following edges as deep as possible before returning to the next edge



Graph Traversal

- Depth First Search
  - Visit nodes following edges as deep as possible before returning to the next edge



- DFS Outputs
  - Spanning Tree (DAG)
  - Preordered Vertices
  - Postordered Vertices
  - Reverse Postorder Vertices

Reaching Definition Analysis

- A variable assignment $[x := a]^{\ell}$ may reach a code location if there is an execution of the program where x was last assigned at ℓ when the code location is reached

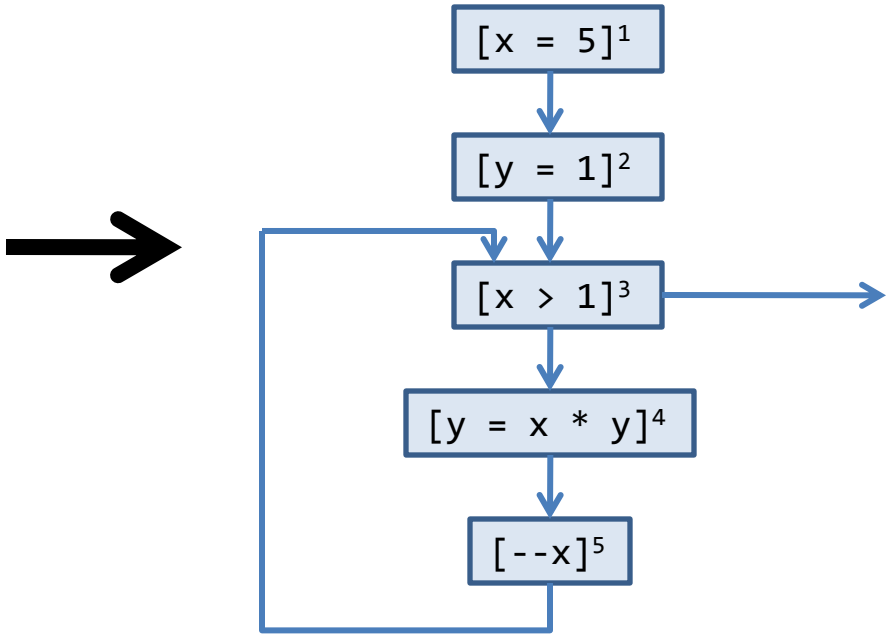
- An assignment reaches the entry of a block if it reaches the exit of any of the blocks that precede it

Reaching Definition Analysis

```
x = 5;
y = 1;
while (x > 1)
{
    y = x * y;
    --x;
}
```

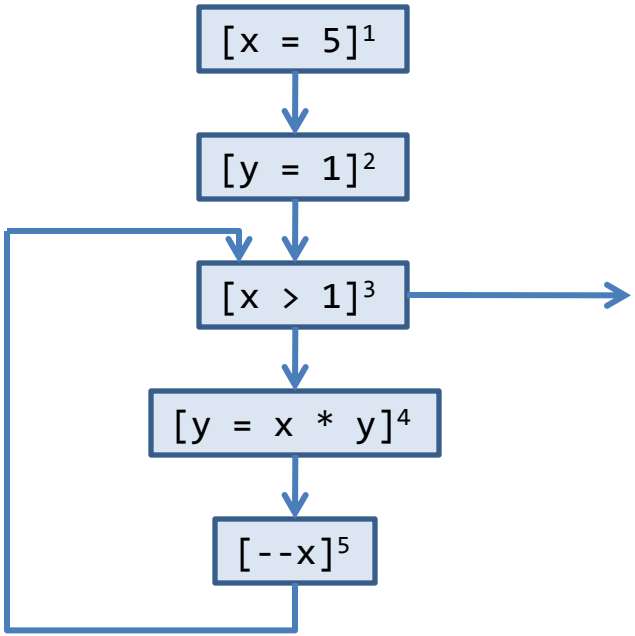
Reaching Definition Analysis

```
x = 5;
y = 1;
while (x > 1)
{
    y = x * y;
    --x;
}
```



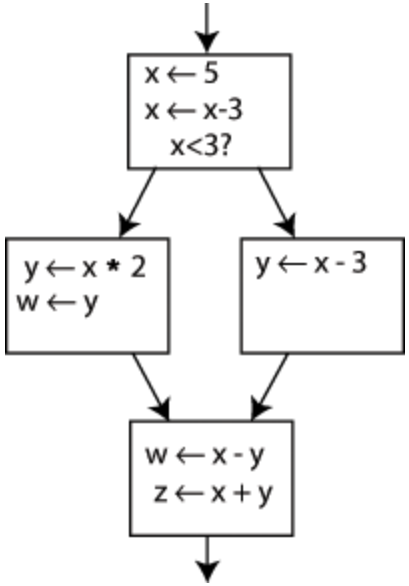
Reaching Definition Analysis

	RD_{entry}	RD_{exit}
1	(x, ?), (y, ?)	(x, 1), (y, ?)
2	(x, 1), (y, ?)	(x, 1), (y, 2)
3	(x, 1), (x, 5), (y, 2)	(x, 1), (x, 5), (y, 2)
4	(x, 1), (x, 5) (y, 2), (y, 4)	(x, 1), (x, 5) (y, 4)
5	(x, 1), (x, 5), (y, 4)	(x, 5), (y, 4)



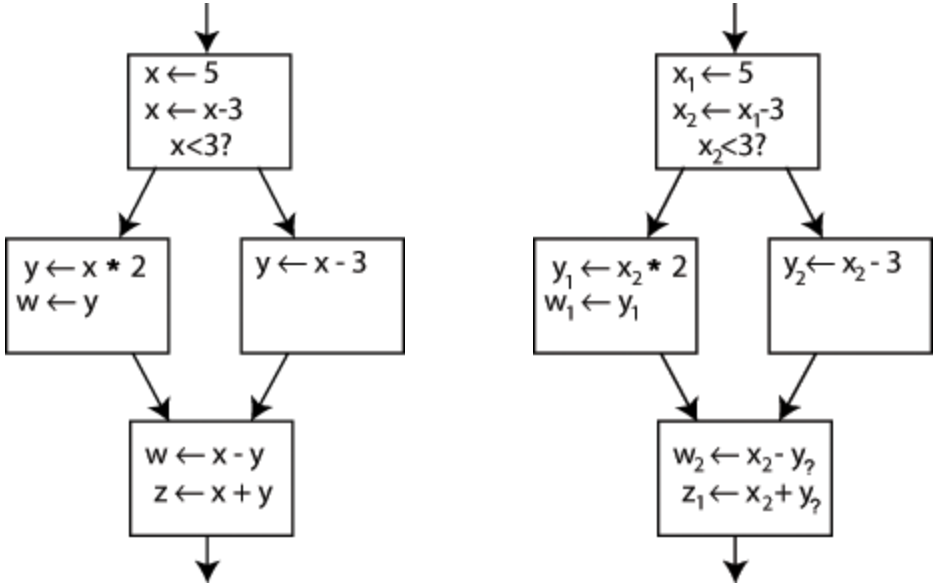
Single Static Assignment

- Intermediate form used by several compilers in which every variable is assigned only once



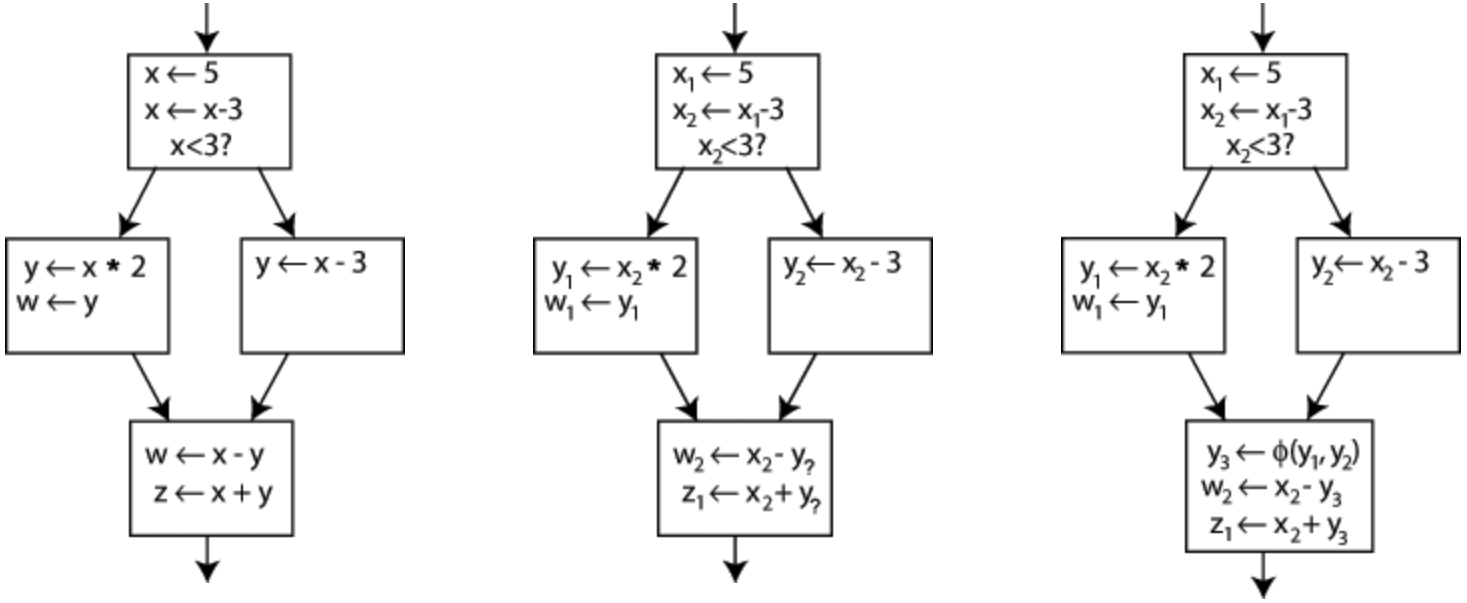
Single Static Assignment

- Use-definition relationships explicit
  - Each use reached by only one definition
  - Each definition dominates all uses



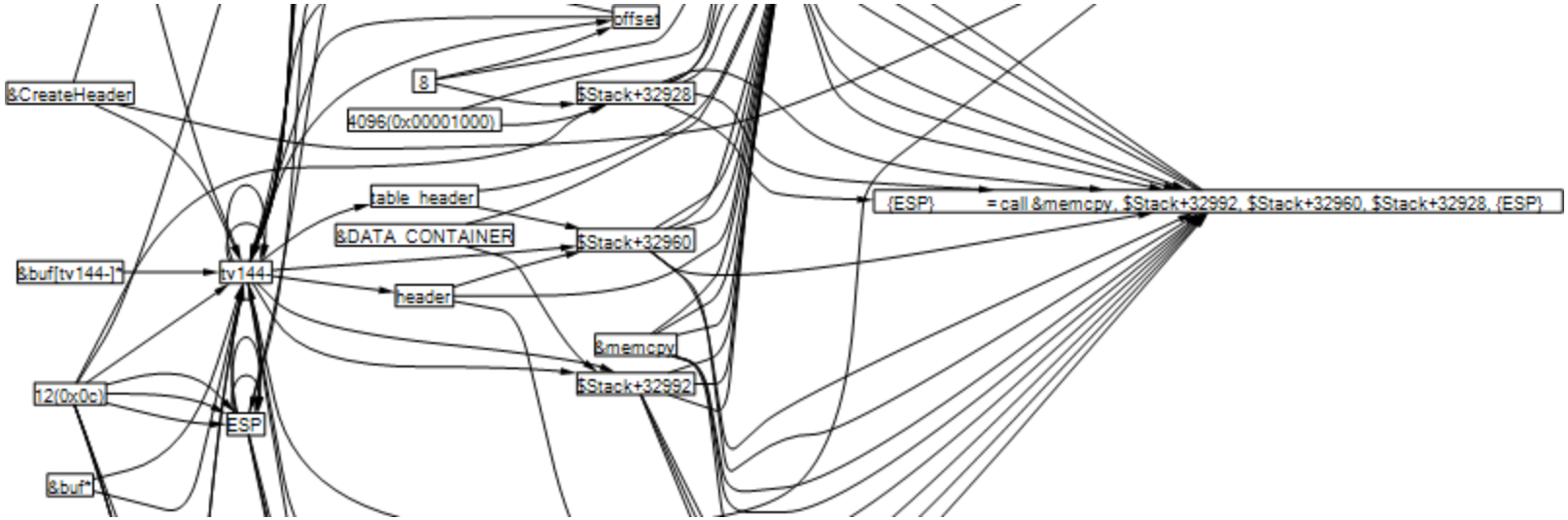
Single Static Assignment

- Special Φ (Phi) instructions are added to the beginning of blocks to represent joins of different versions of the same variable



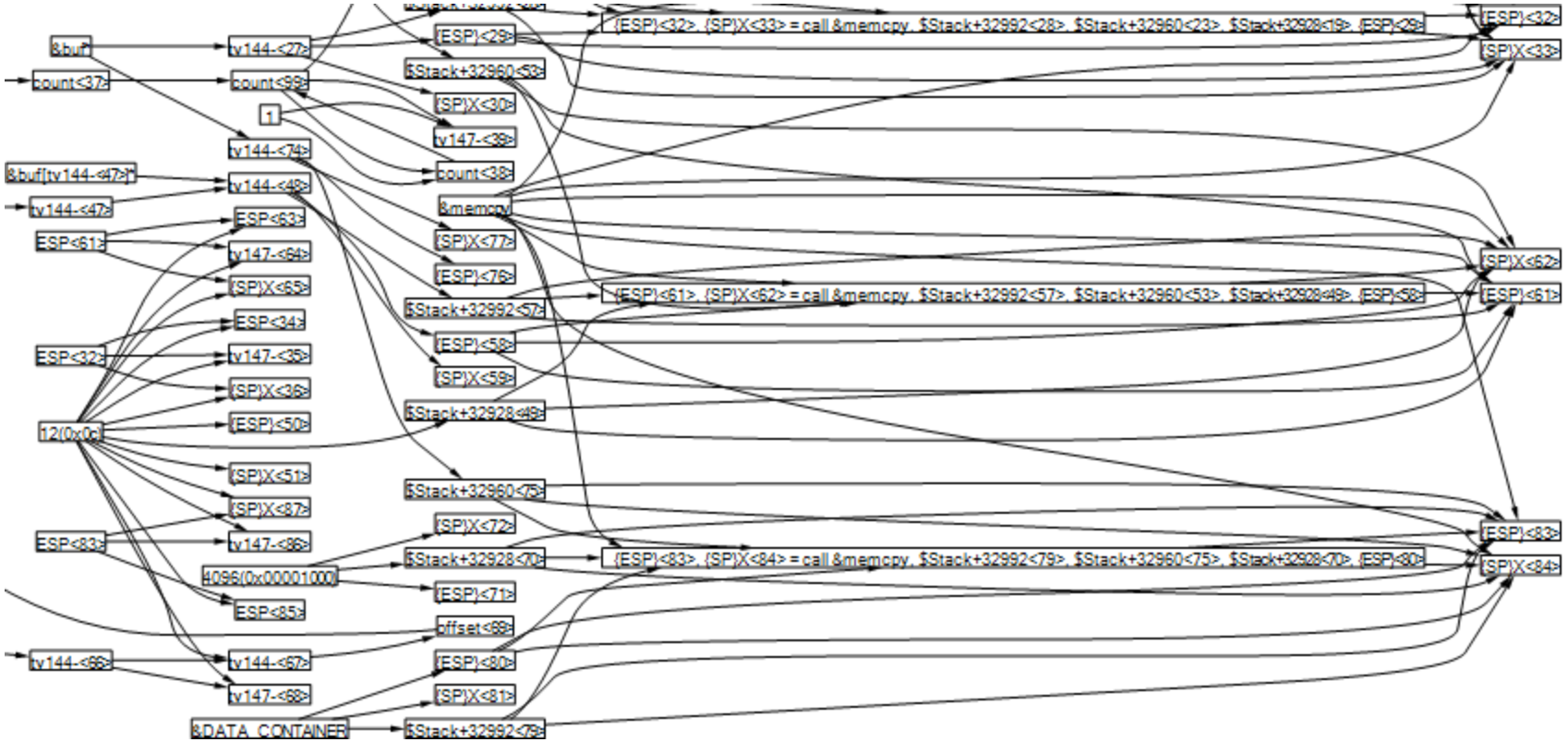
Data Flow Graphs

- Reaching definitions allow the construction of a context-free data flow graph



Data Flow Graphs

- Converting to SSA form allows the simple construction of a contextual data flow graph



Applied Data Flow Analysis

- API Path Validation
  - Determine whether there is a code path from data input function to a function using the data that does not flow through a sanitizing function
  - Method
    - Create an array of bit vectors to hold each path to each function
    - Propagate inherited bit vectors by performing a union on the two bit vectors
  - Real world use – SQL Injection prevention

Applied Data Flow Analysis

- Syntax Model Inference
  - Determine the type layout of every abstract structure that reaches a specified function call
  - Method
    - Calculate call graph for target function
    - Gather Reaching Definition data for all functions in graph
    - Record type for each definition in each function
    - Walk unique call graph paths backwards collecting type flow information
  - Real world use – generate fuzzer definitions

Applied Data Flow Analysis

- Integer Overflow Detection
  - Given a call to an allocation function, determine whether the input size could have wrapped
  - Method
    - Trace data input to memory allocation functions backward
    - Determine if the value is generated in with potentially user controlled data
  - Real world use – detect bugs!

Final Thoughts

- Phoenix is amazingly powerful and extensible. It will change how academic compiler research is done on the Windows platform
- The security industry has a lot to learn from the academic archives of the last 30 years. Read Dawson Engler, David Wagner, Cousot
- Improved programming processes and advances in static analysis is and will continue to improve software security

Get Involved!

- Get phoenix
  - http://research.micorosft.com/phoenix
- Participate in online phoenix forums
- Contact us at switech@microsoft.com