Specification-Based Analysis and Enforcement

Jonathon Giffin, Somesh Jha, Barton Miller
University of Wisconsin

Overview

- Intrusion detection and specification-based monitoring
- An unusual intrusion path
  - The Condor attack: How to easily do dangerous and malicious things to a running job
- How to detect attempted intrusions with pre-execution static analysis and runtime monitoring
- Precision & performance results for 3 programs
- Recent work
  - Null call insertion to improve precision & performance
  - Analysis of shared objects

Intrusion Detection

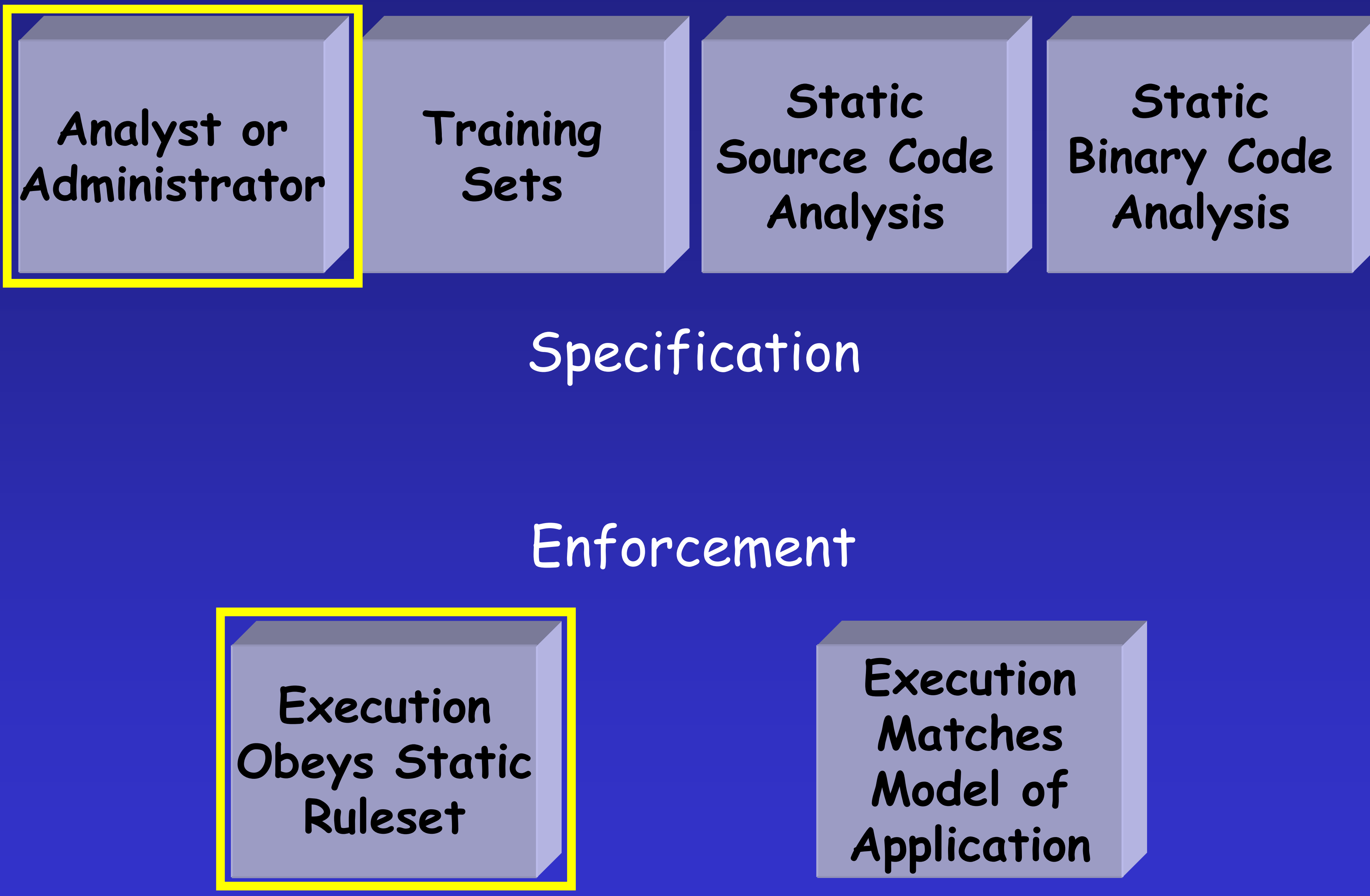
Goal: Discover attempts to maliciously gain access to a system

Goal: Discover attempts to maliciously gain access to a system

Misuse Detection	Specification-Based Monitoring	Anomaly Detection
• Specify patterns of attack or misuse	• Specify constraints upon program behavior	• Learn typical behavior of application
• Ensure misuse patterns do not arise at runtime	• Ensure execution does not violate specification	• Variations indicate potential intrusions
• Snort	• Our work; Ko, et. al.	• IDES
• Rigid: cannot adapt to novel attacks	• Specifications can be cumbersome to create	• High false alarm rate

Specification-Based Monitoring

- Two components:
  - Specification: Indicates constraints upon program behavior
  - Enforcement: How the specification is verified at runtime or from audit data


Analyst or Administrator
Training Sets
Static Source Code Analysis
Static Binary Code Analysis
Specification
Enforcement
Execution Obeys Static Ruleset
Execution Matches Model of Application

Representative Work by Ko, et. al.

- Specification: Programmers or administrators specify correct program behavior

```
PROGRAM fingerd
    read(X) :- worldreadable(X);
    bind(79);
    write("/etc/log");
    exec("/usr/ucb/finger");
END
```

- Enforcement: At runtime, only allow actions that match the specified policy

Analyst or Administrator

Training Sets

Static Source Code Analysis

Static Binary Code Analysis

Specification

Enforcement

Execution Obeys Static Ruleset

Execution Matches Model of Application

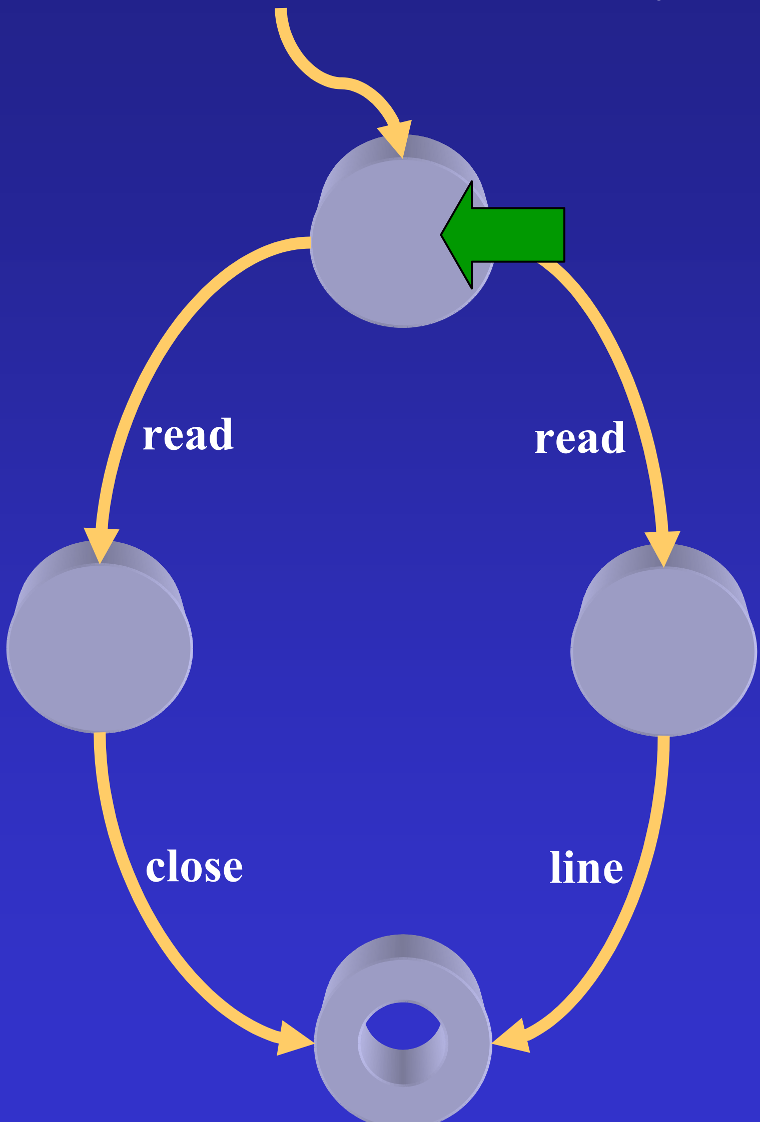
Our Approach

```
function:
  save %sp, 0x96, %sp
  cmp %i0, 0
  bge L1
  mov 15, %o1
  call read
  mov 0, %o0
  call line
  nop
  b L2
  nop
L1:
  call read
  mov %i0, %o0
  call close
  mov %i0, %o0
L2:
  ret
  restore
```

Specification: Static analysis of binary code

- Specifications are automatically generated
- Not reliant upon programmers to produce accurate specifications
- Analyzes all execution paths
- Source code may be unavailable

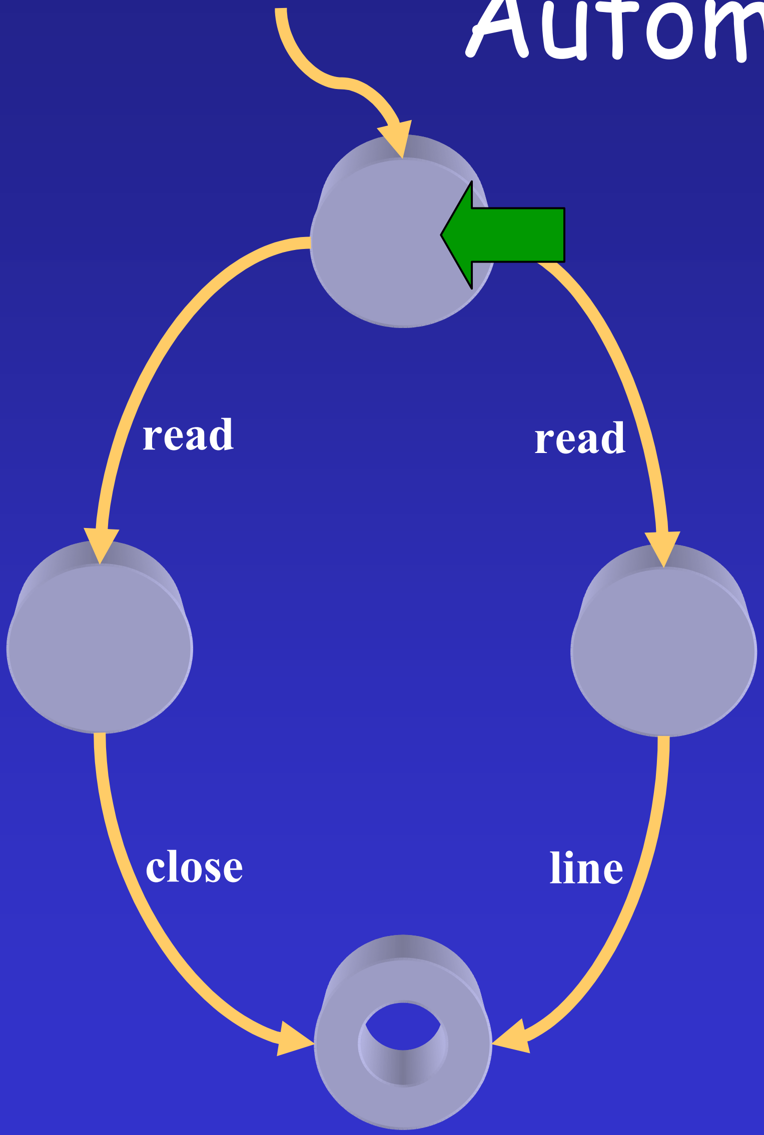
Our Approach



Enforcement: Operate an automaton modeling correct system call sequences

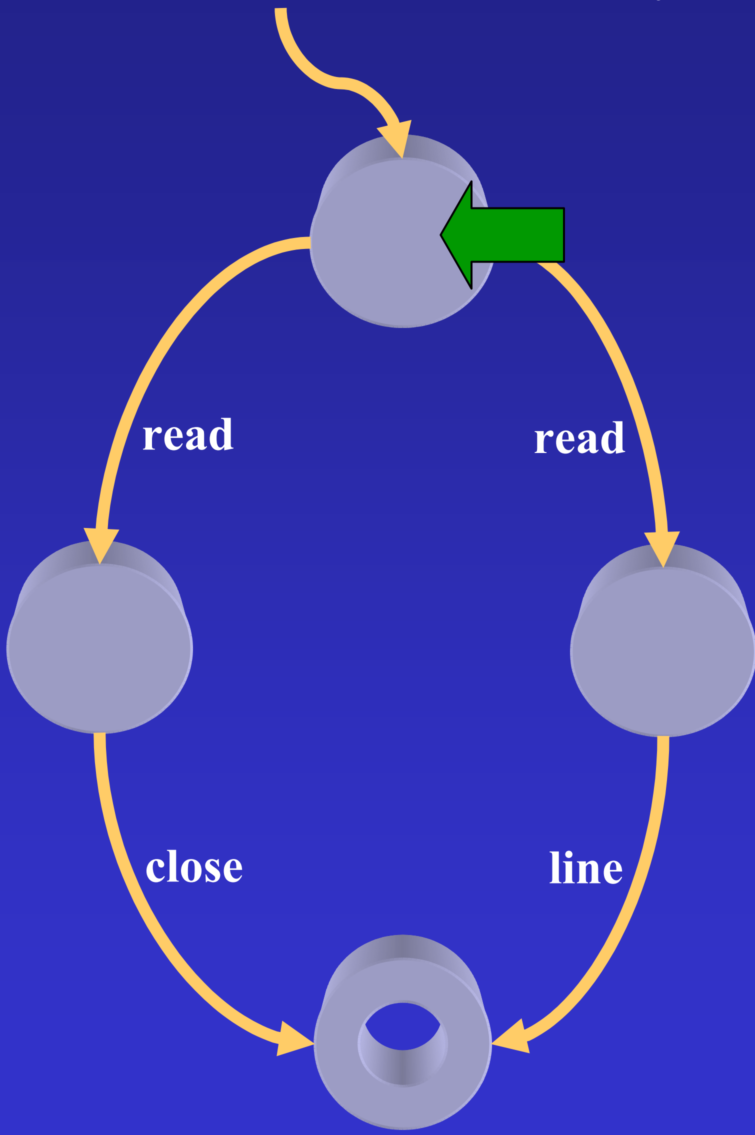
- Dynamic ruleset
- More expressive than static ruleset of Ko, et. al.

Non-Deterministic Finite Automaton (NFA)



- Structure
  - States
  - Labeled edges between states
- Edge labels are input symbols – call names
- Path to any accepting state defines valid sequence of calls

Our Approach

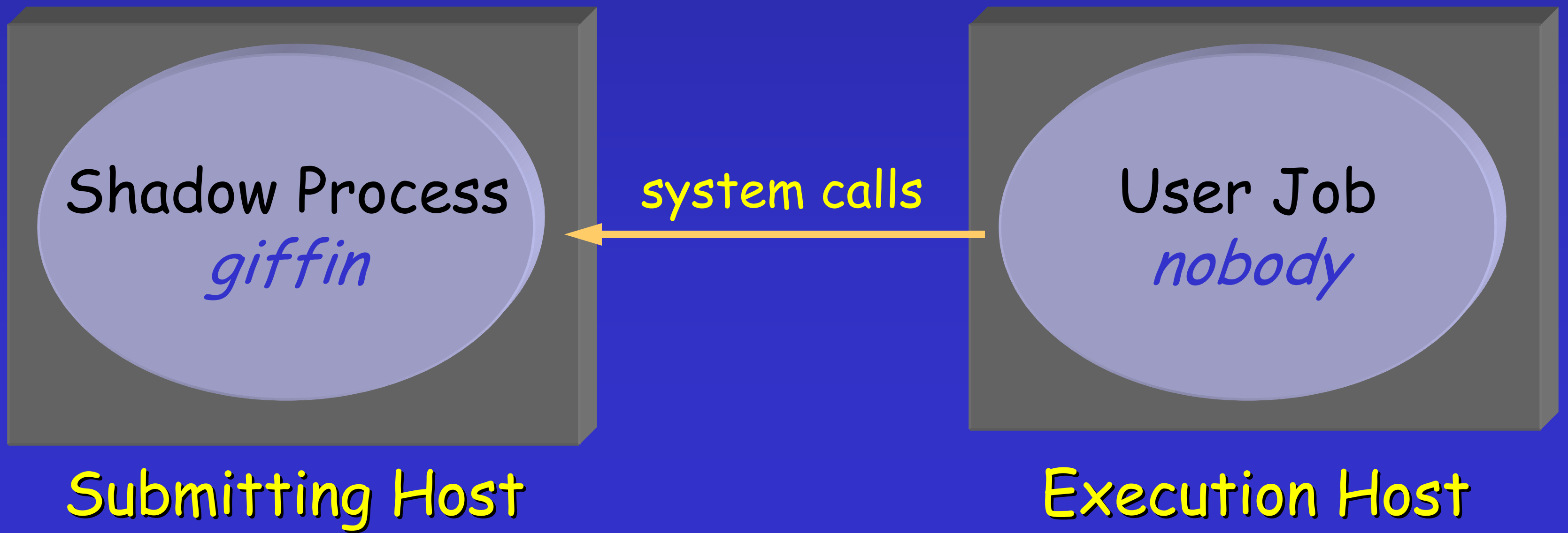


Enforcement: Operate an automaton modeling correct system call sequences

- Dynamic ruleset
- More expressive than static ruleset of Ko, et. al.

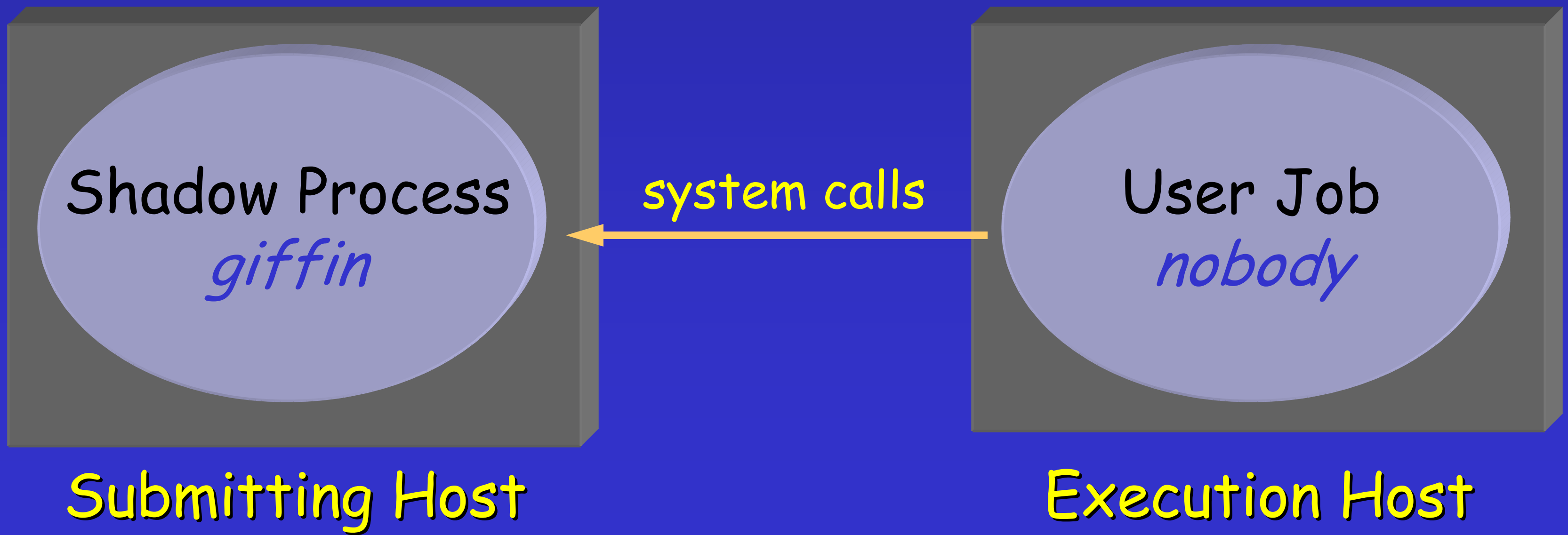
Example: The Condor Attack

- Users dispatch programs for remote execution
- Remote jobs send critical system calls back to local machine for execution



Example: The Condor Attack

- Attackers can manipulate remotely executing program to gain access to user's machine



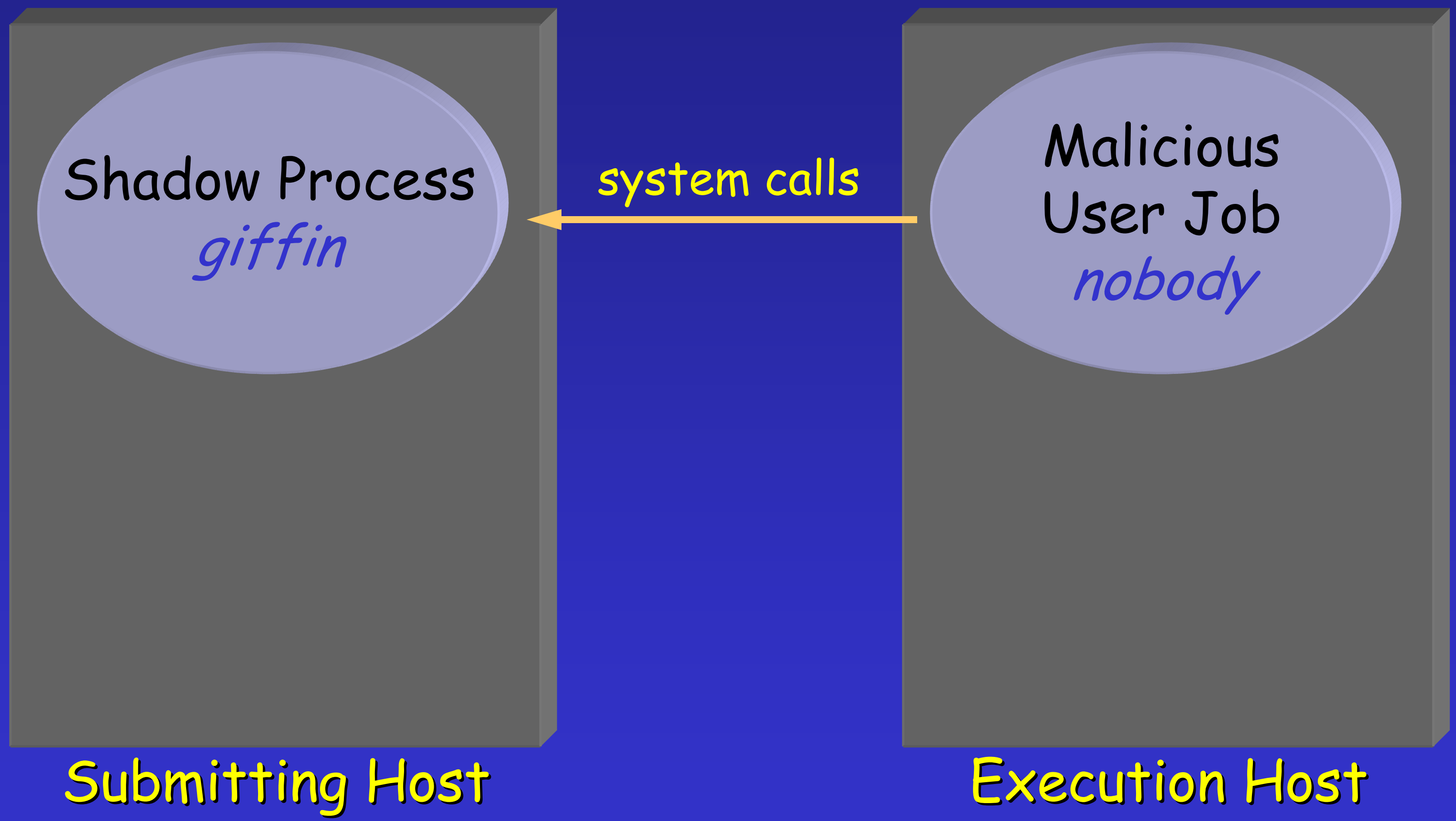
A New View

- Running programs are objects to be easily manipulated
- The vehicle: the DynInst API

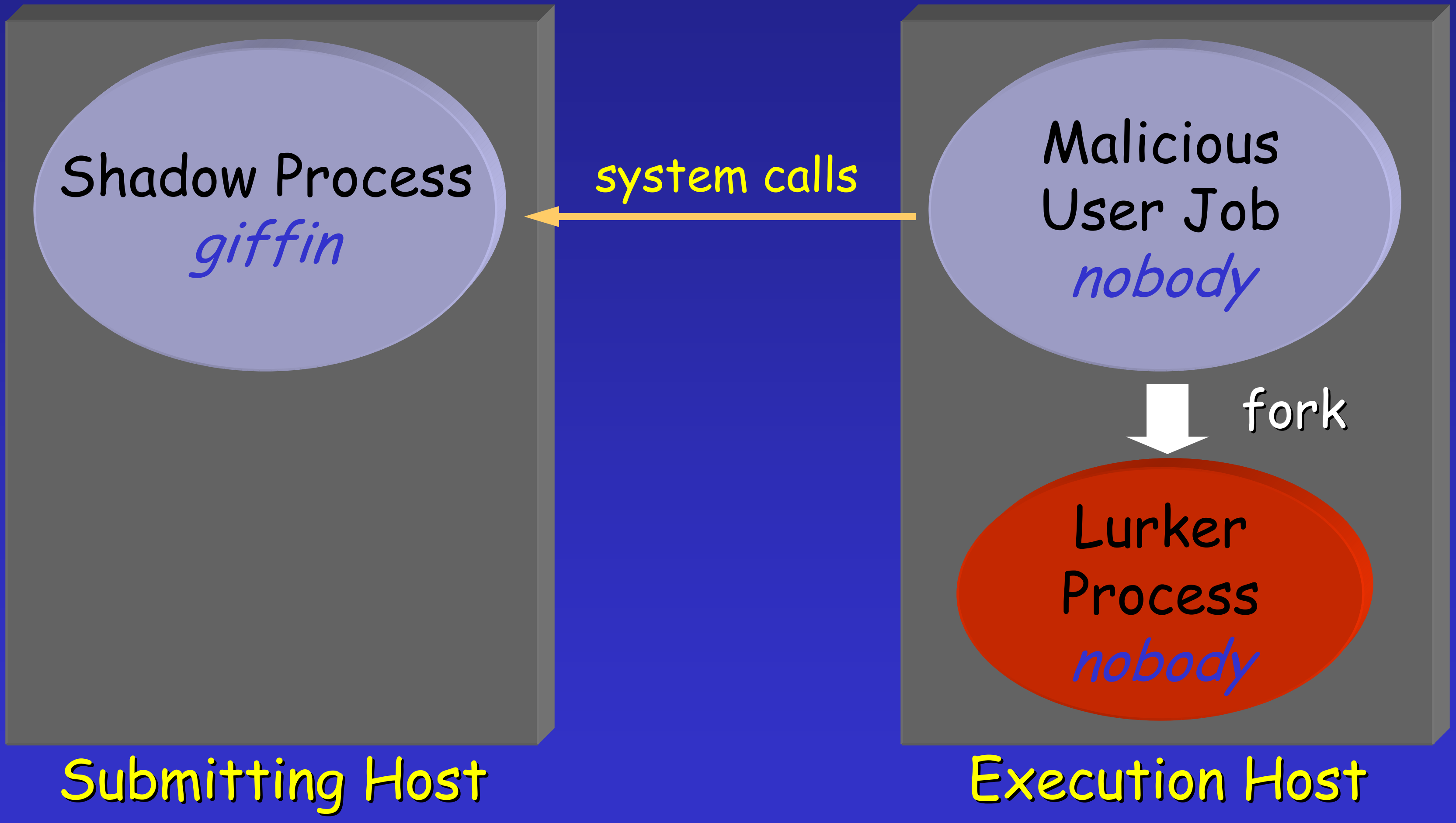
DynInst: Dynamic Instrumentation

- Machine independent library for instrumentation of running processes
- Modify control flow of the process:
  - Load new code into the process
  - Remove, replace, or redirect function calls
  - Asynchronously call any function in the process

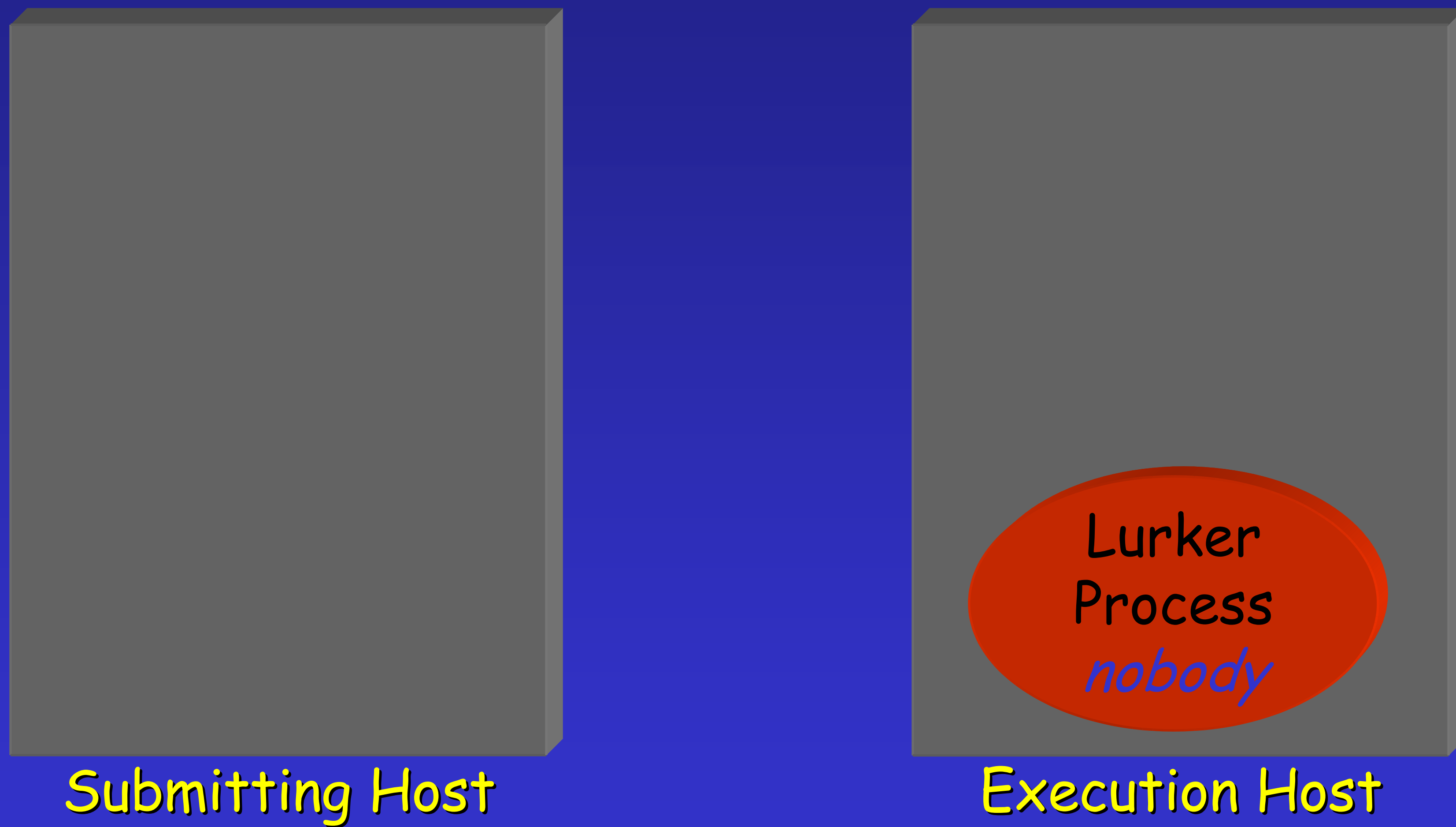
Condor Attack: Lurking Jobs



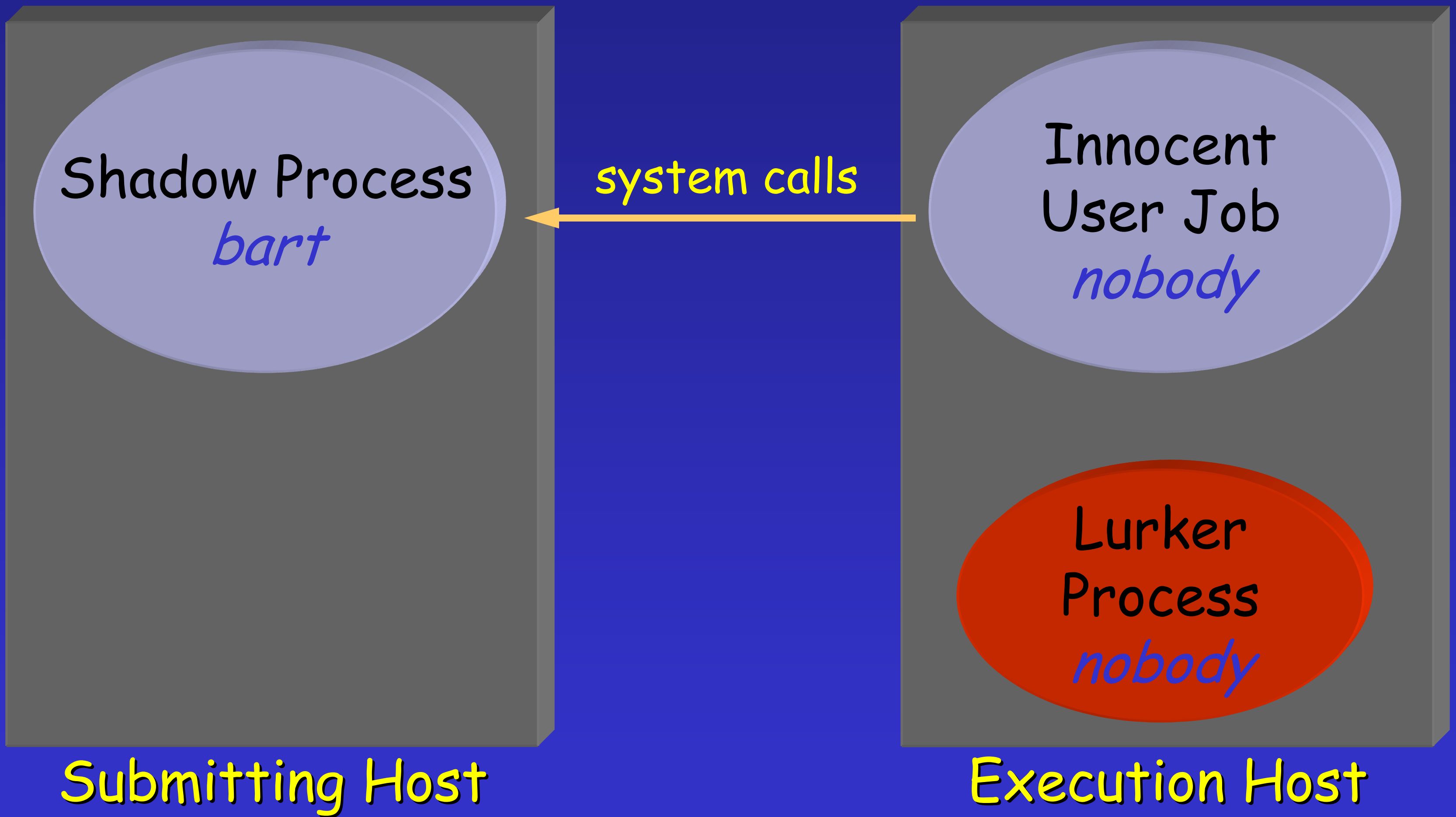
Condor Attack: Lurking Jobs



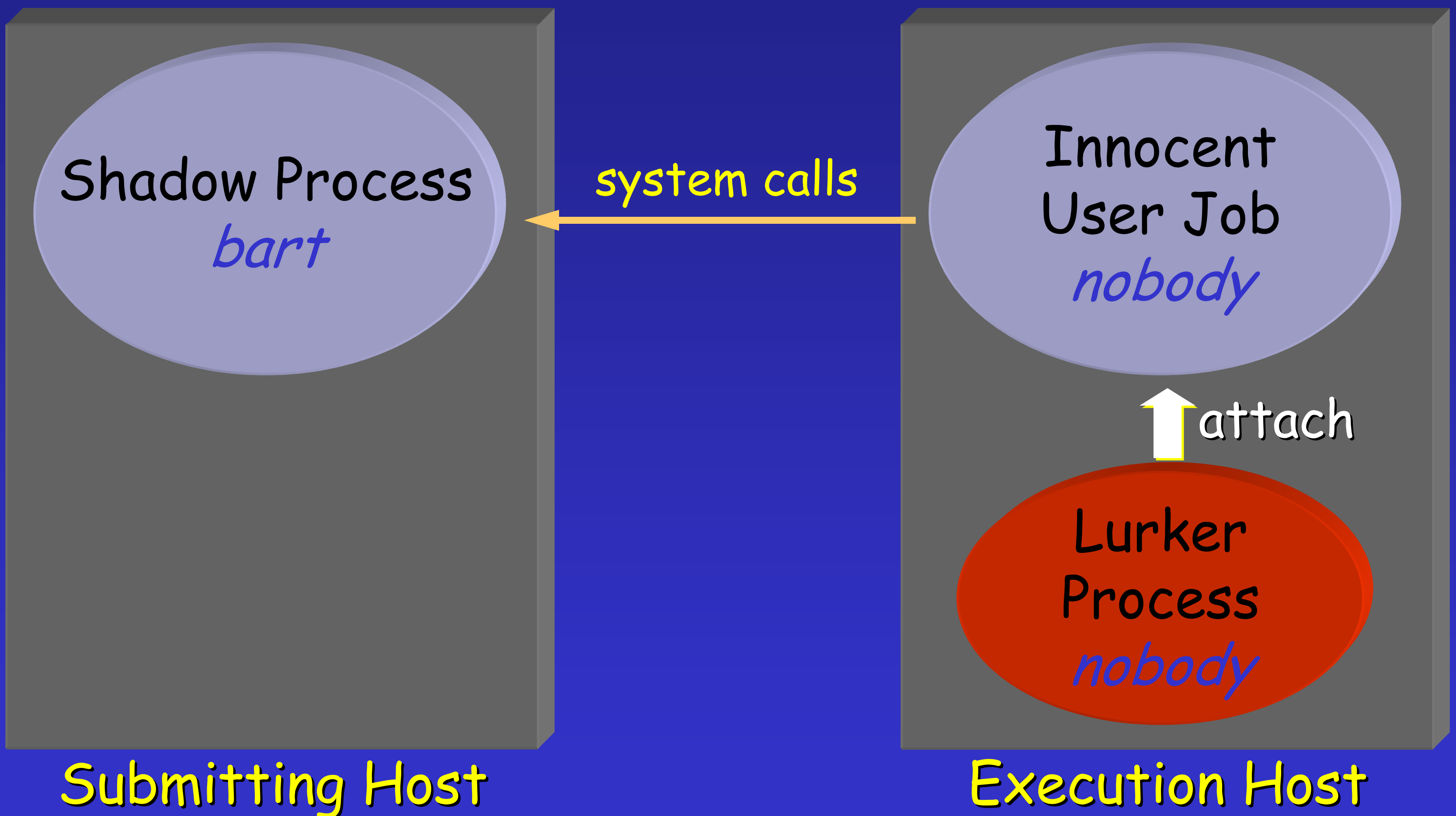
Condor Attack: Lurking Jobs



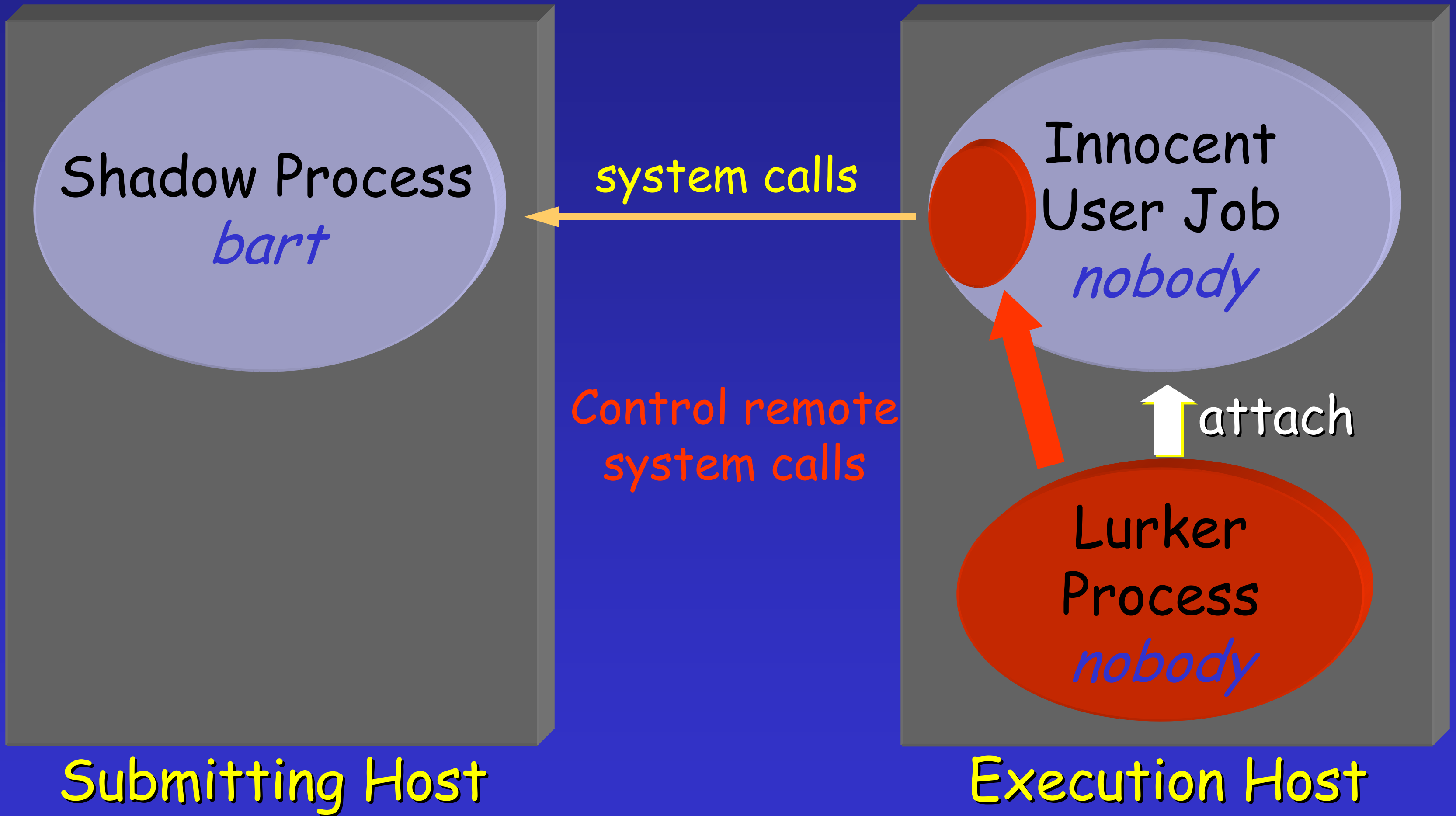
Condor Attack: Lurking Jobs



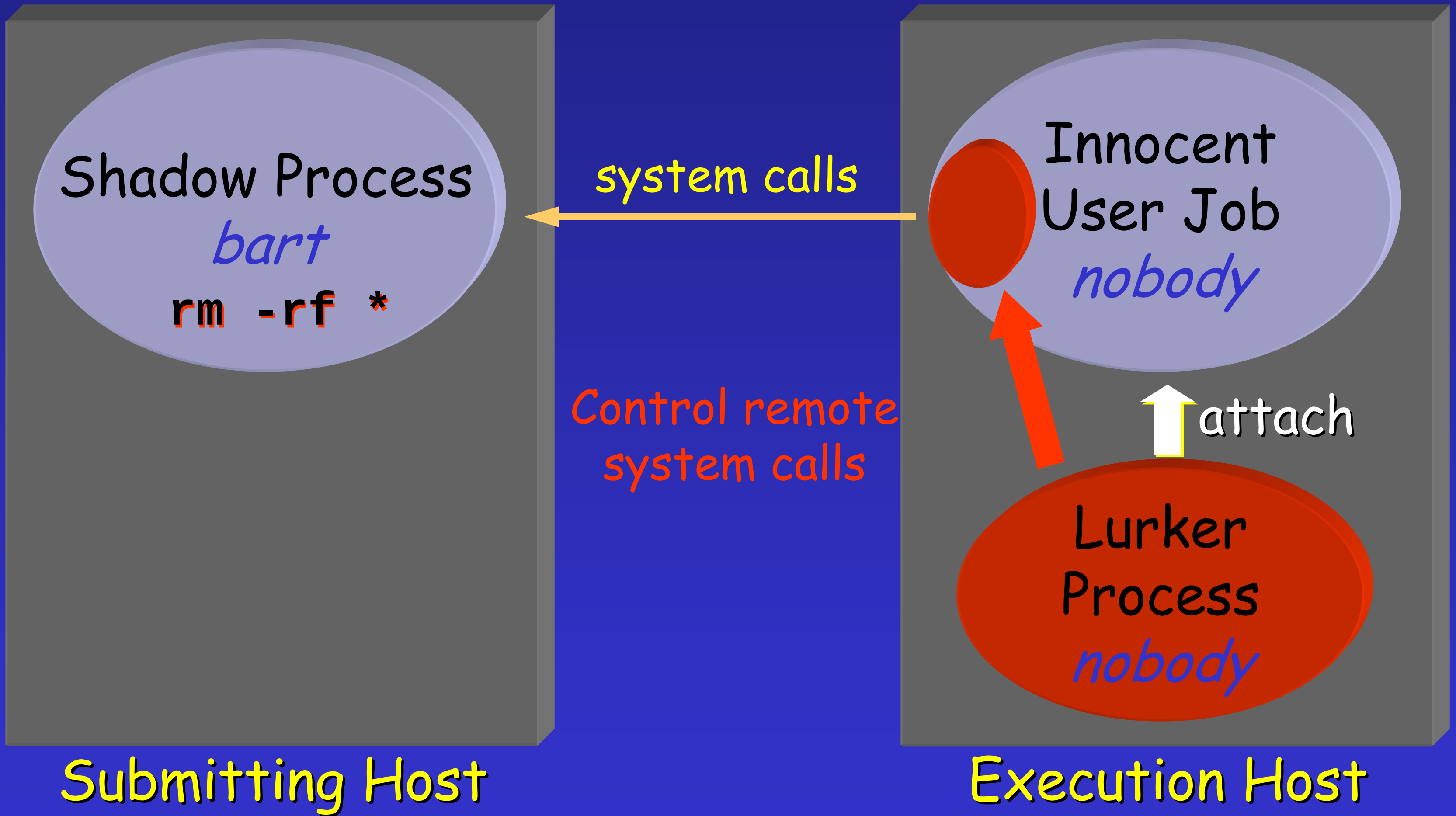
Condor Attack: Lurking Jobs



Condor Attack: Lurking Jobs



Condor Attack: Lurking Jobs



Can We Safely Execute Our Jobs Remotely?

The threats:

1. Cause the job to make improper remote system calls.
2. Cause the job to calculate an incorrect answer.
3. Steal data from the remote job.

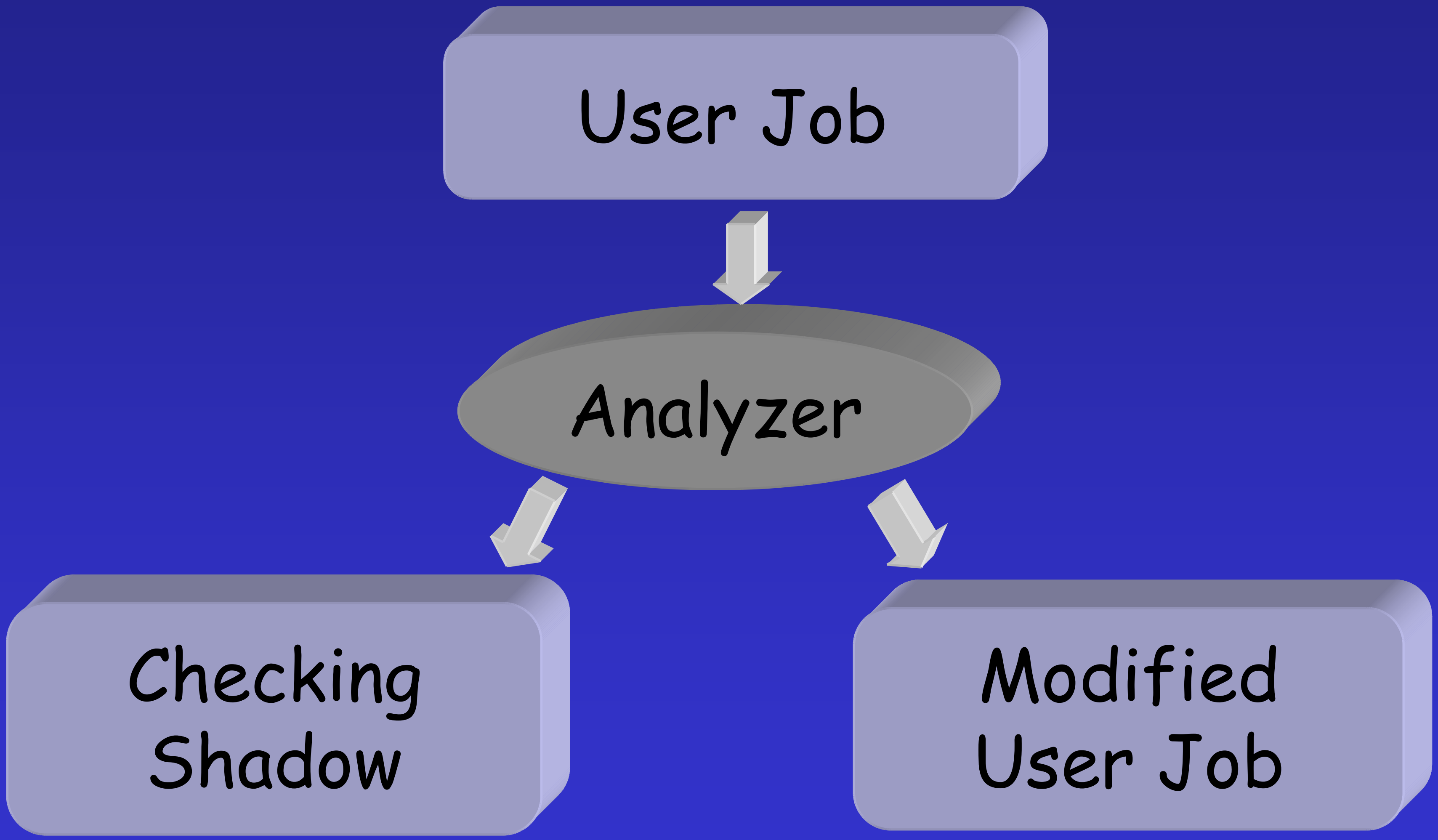
Threat protection strategies:

- Monitor execution of remote job (threat #1)
- File or system call sand-boxing (#1)
- Obfuscate or encode remote job (#1, #3)
- Replicate remote job (#2)

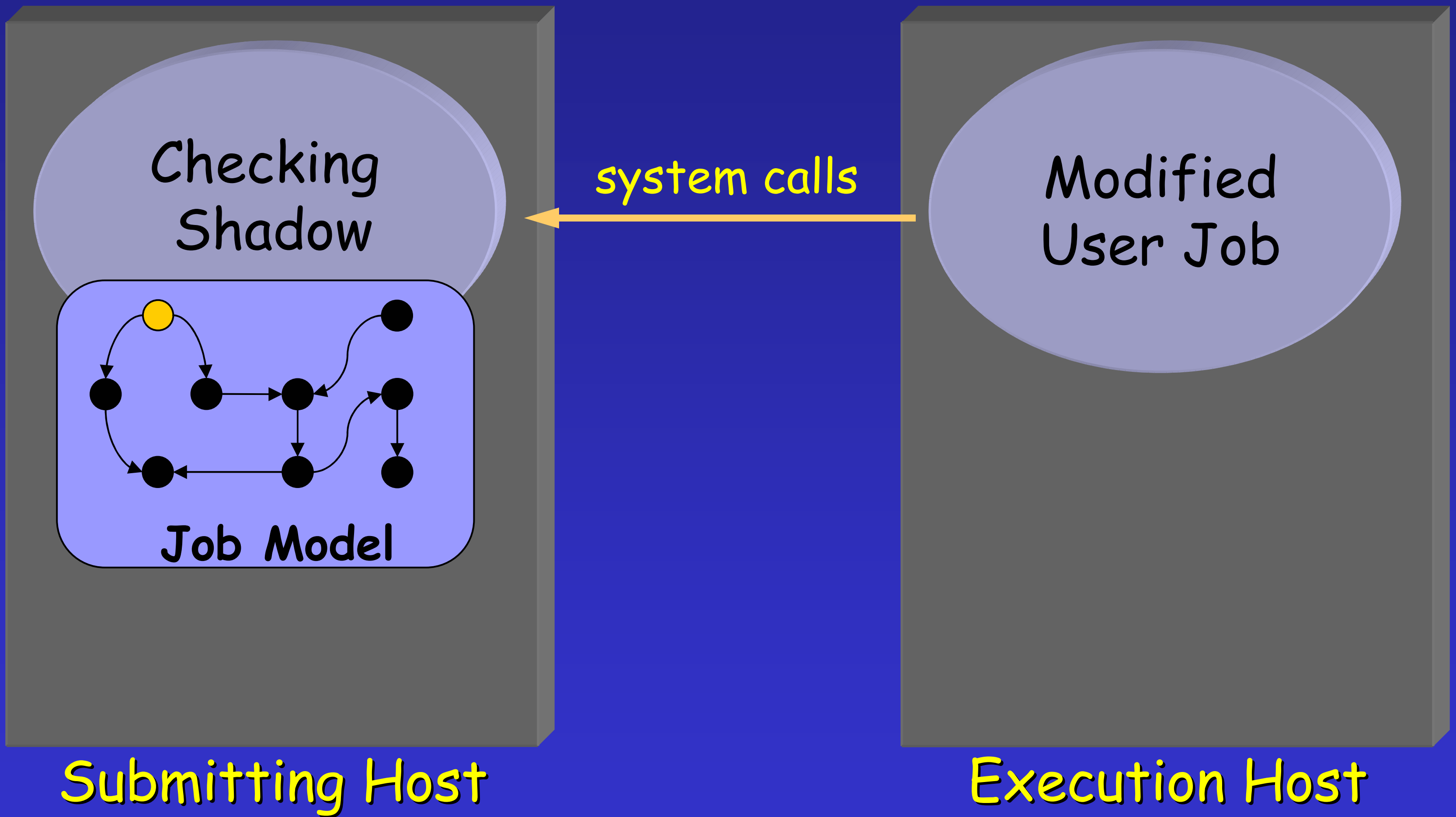
Countering Remote Attacks

- Goal: Even if an intruder can see, examine, and fully control the remote job, no harm can come to the local machine.
- Method: Model all possible sequences of remote system calls. At runtime, update the model with each received call.
- Key technology: Static analysis of binary code.

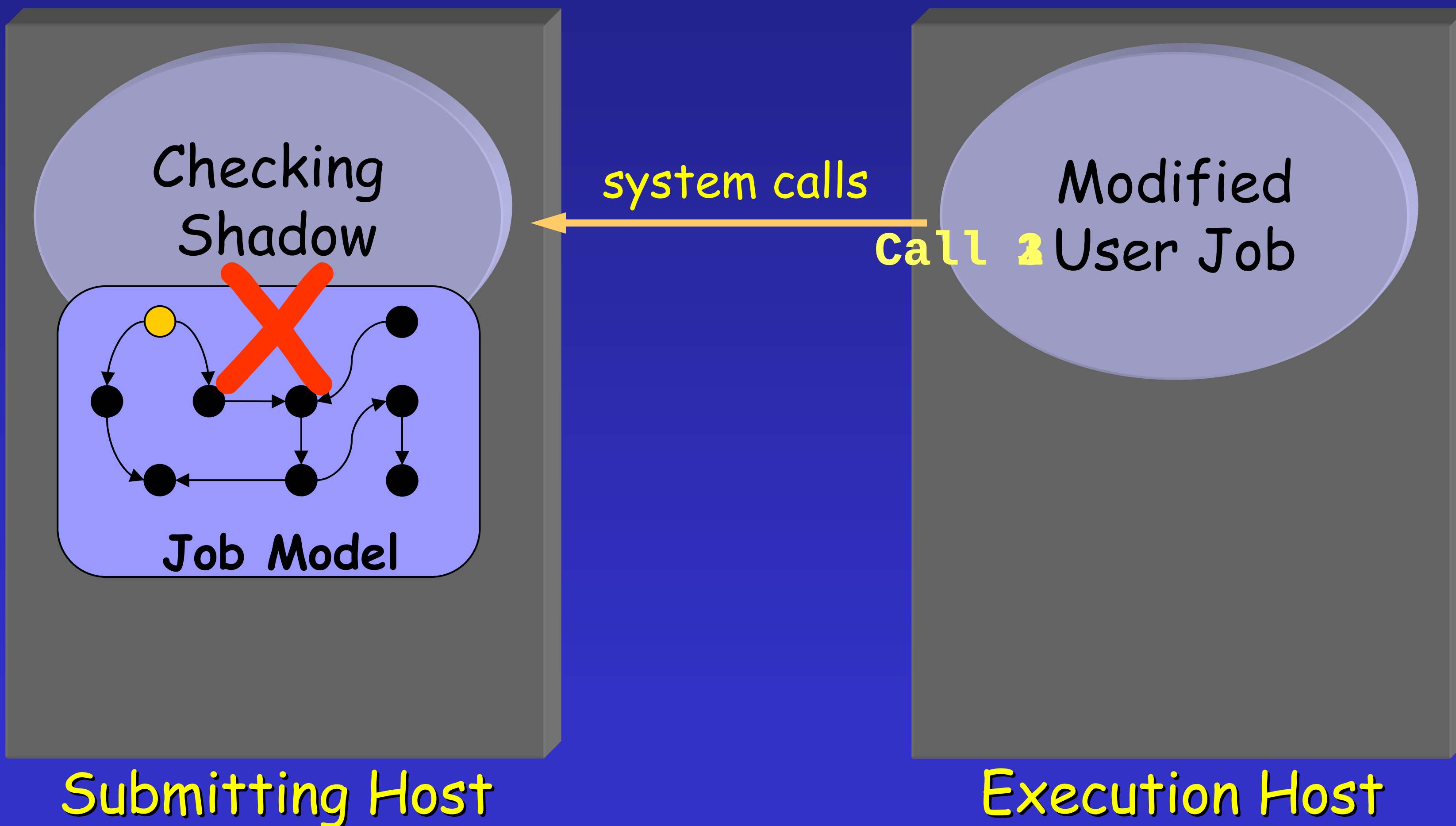
Execution Monitoring



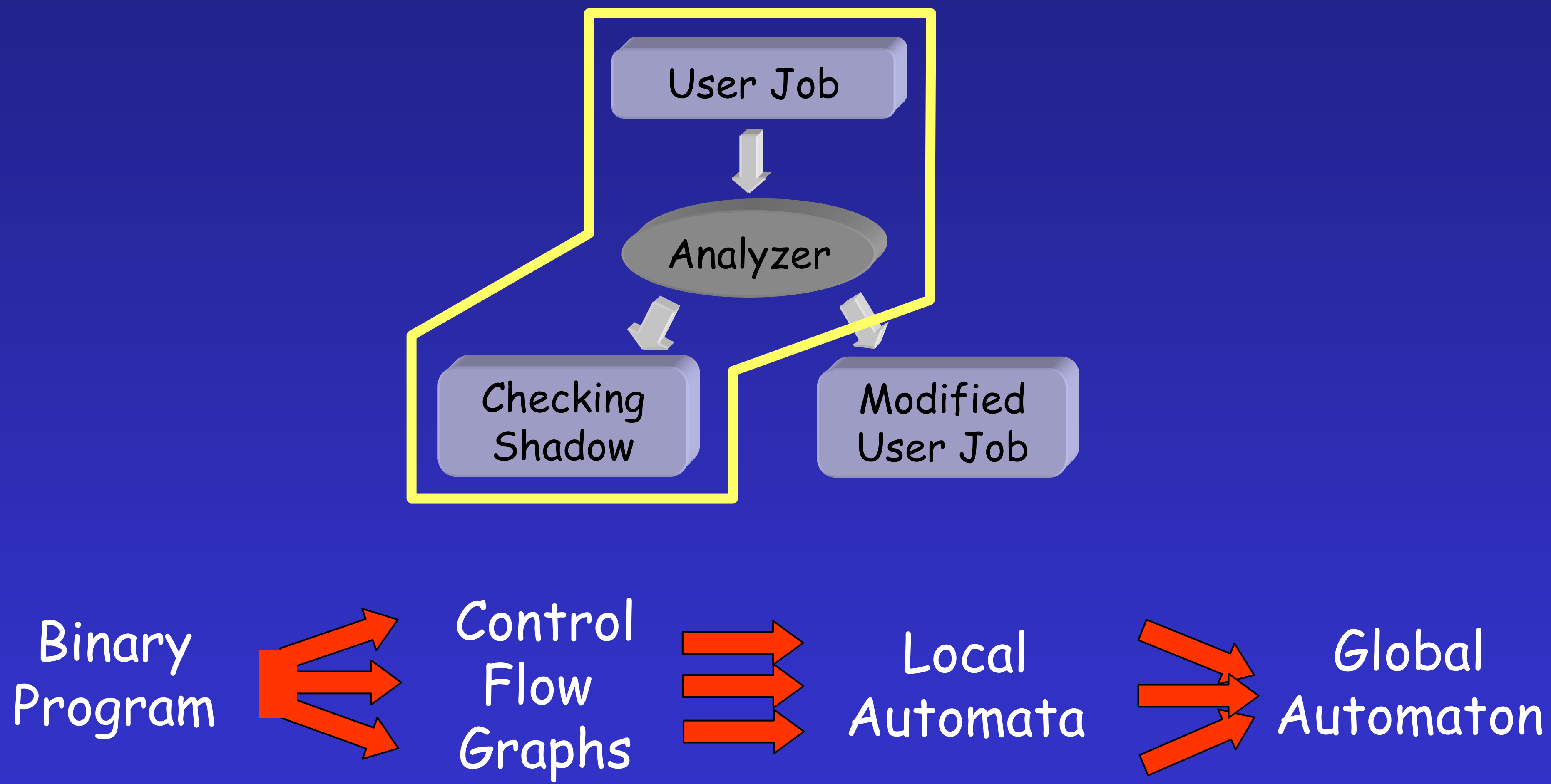
Execution Monitoring



Execution Monitoring

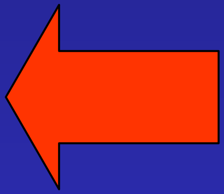


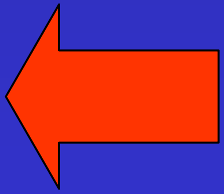
Model Construction



The Binary View (SPARC)

```
function:
  save %sp, 0x96, %sp
  cmp %i0, 0
  bge L1
  mov 15, %o1
  call read
  mov 0, %o0
  call line
  nop
  b L2
  nop
L1:
  call read
  mov %i0, %o0
  call close
  mov %i0, %o0
L2:
  ret
  restore
```

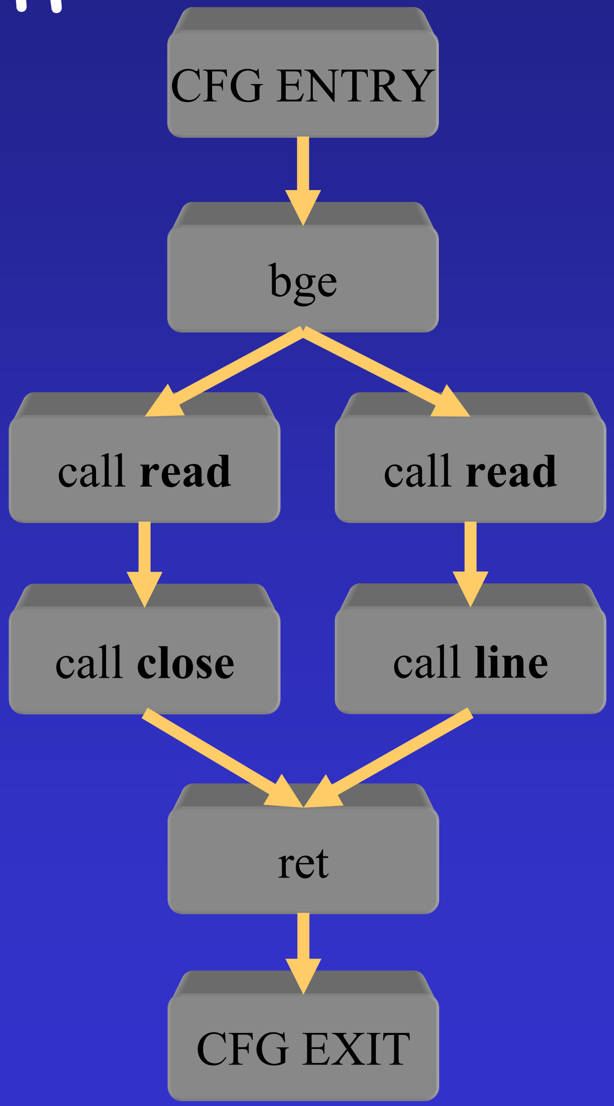




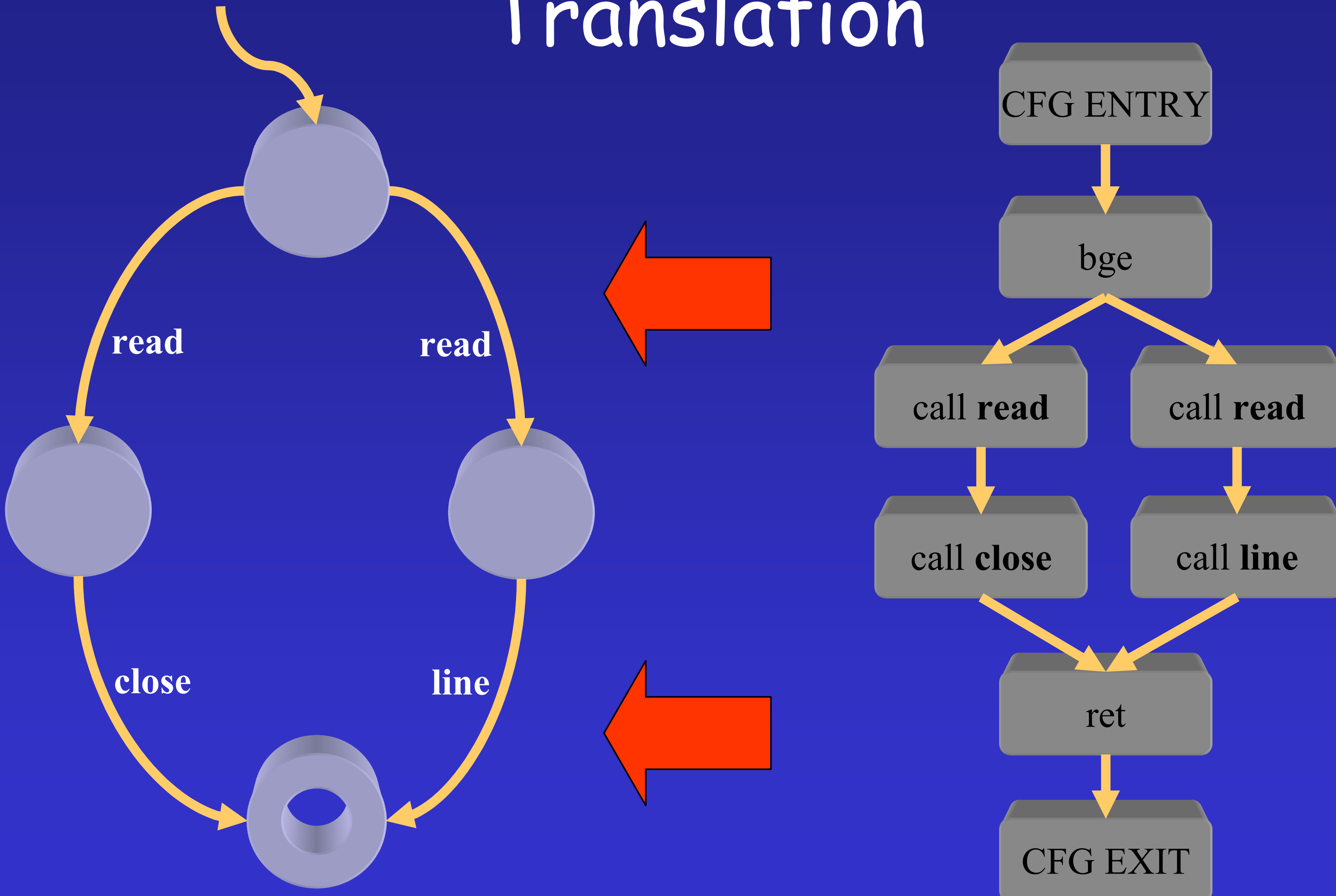
```
function (int a) {
  if (a < 0) {
    read(0, 15);
    line();
  } else {
    read(a, 15);
    close(a);
  }
}
```

Control Flow Graph Generation

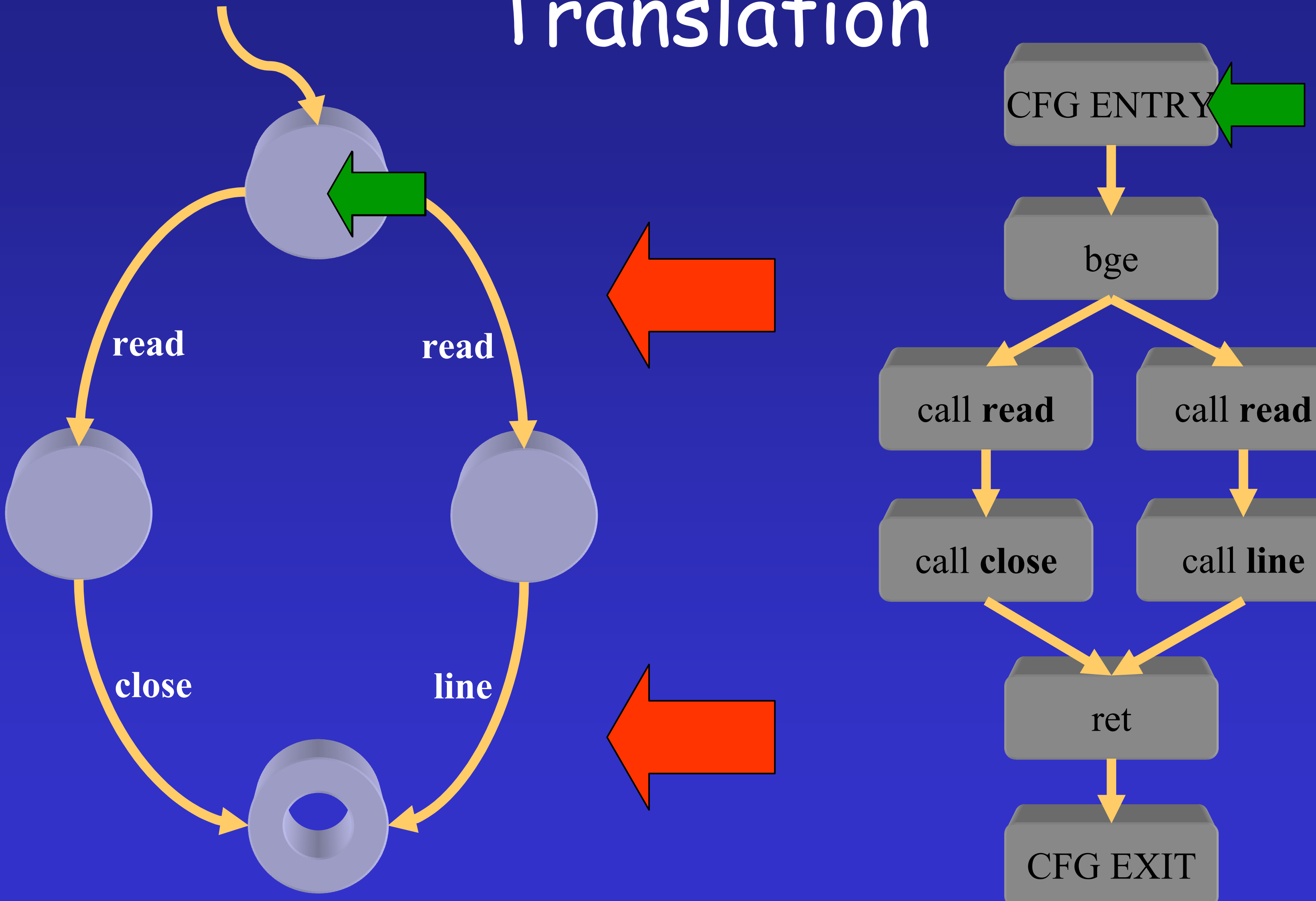
```
function:
  save %sp, 0x96, %sp
  cmp %i0, 0
  bge L1
  mov 15, %o1
  call read
  mov 0, %o0
  call line
  nop
  b L2
  nop
L1:
  call read
  mov %i0, %o0
  call close
  mov %i0, %o0
L2:
  ret
  restore
```



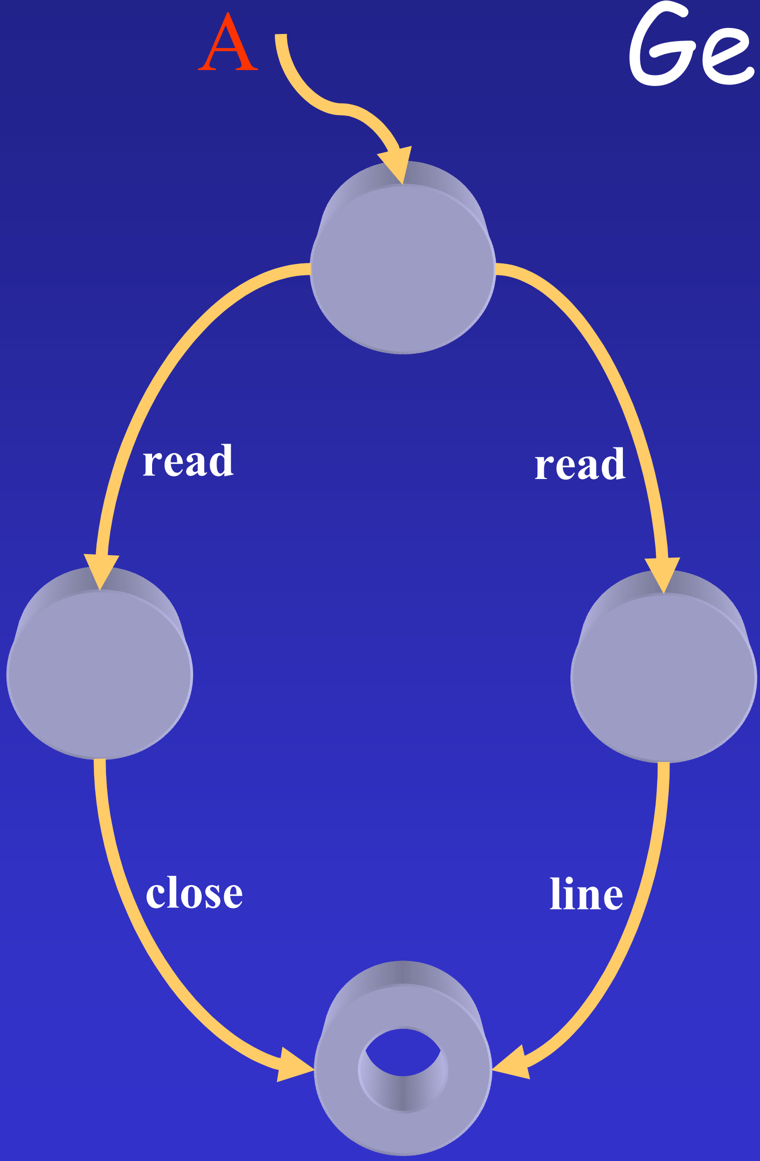
Control Flow Graph Translation



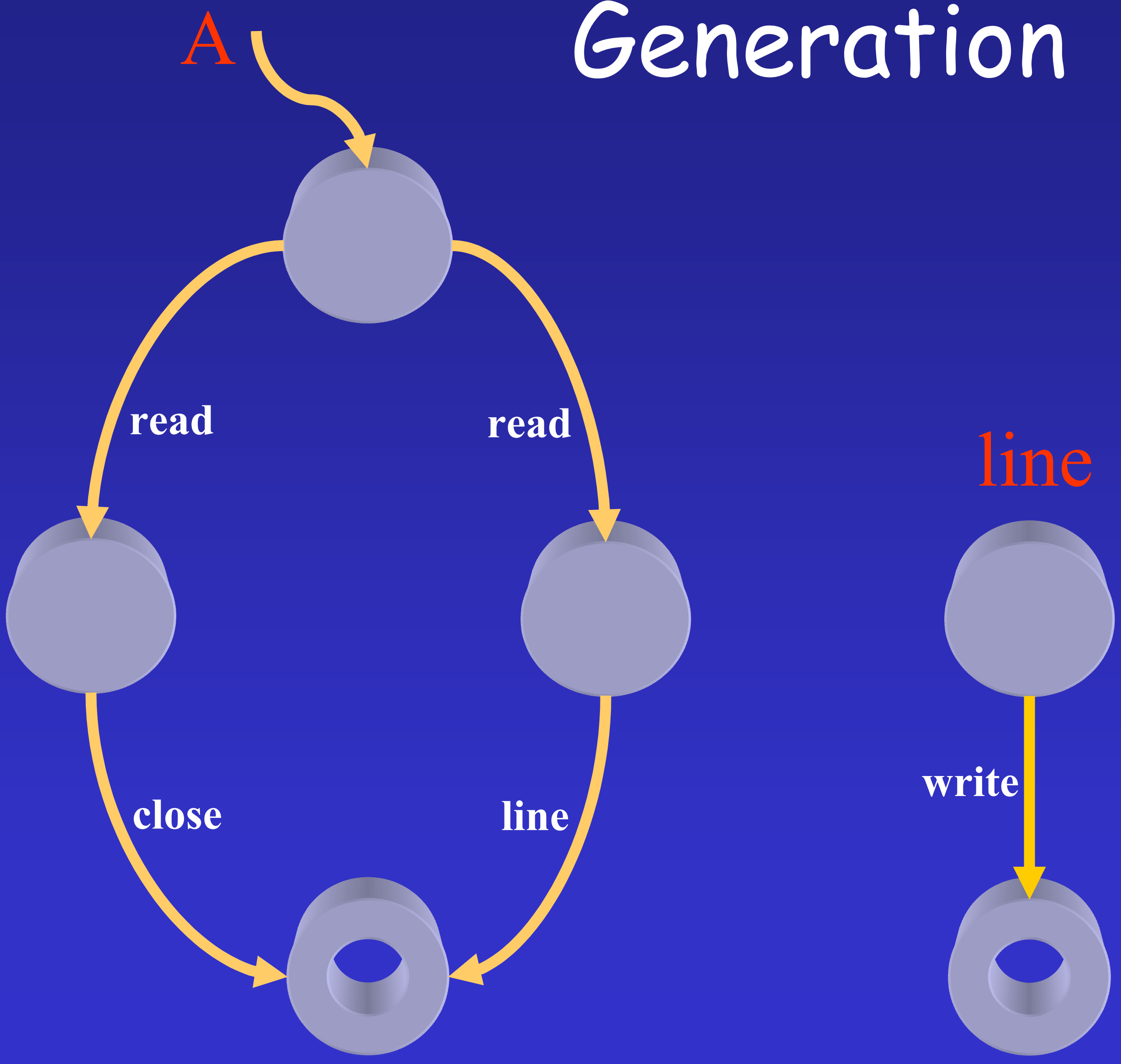
Control Flow Graph Translation



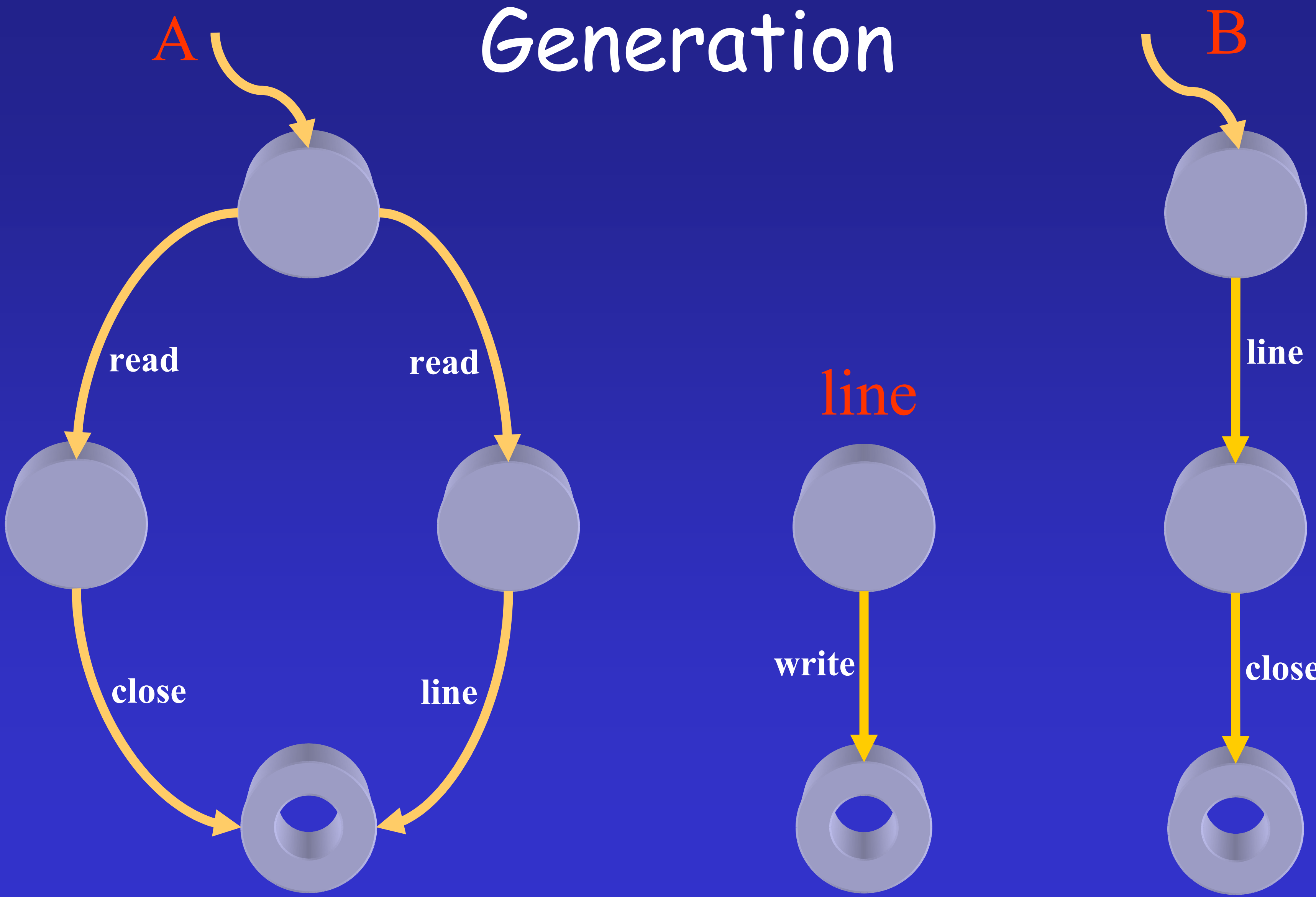
Interprocedural Model Generation



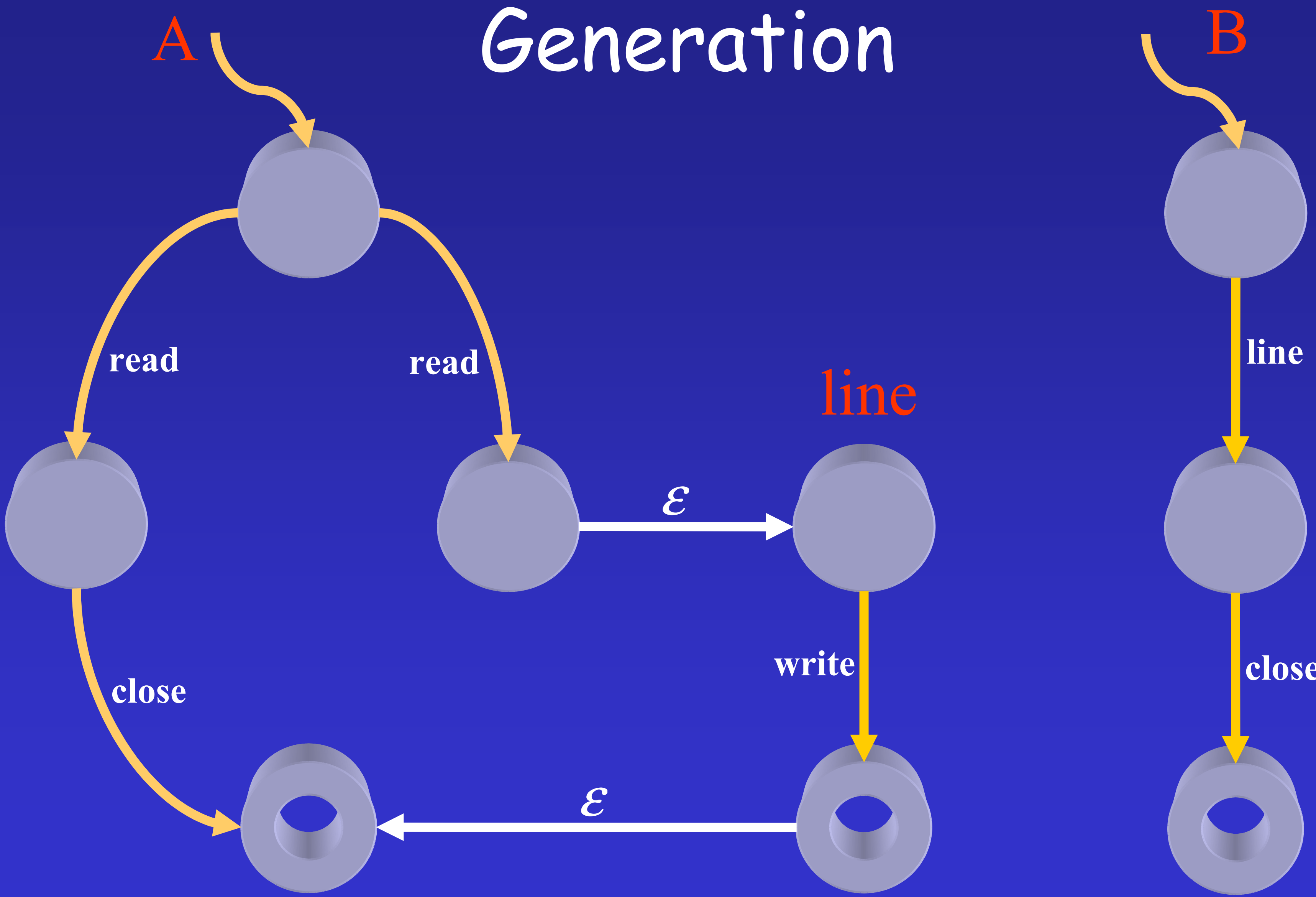
Interprocedural Model Generation



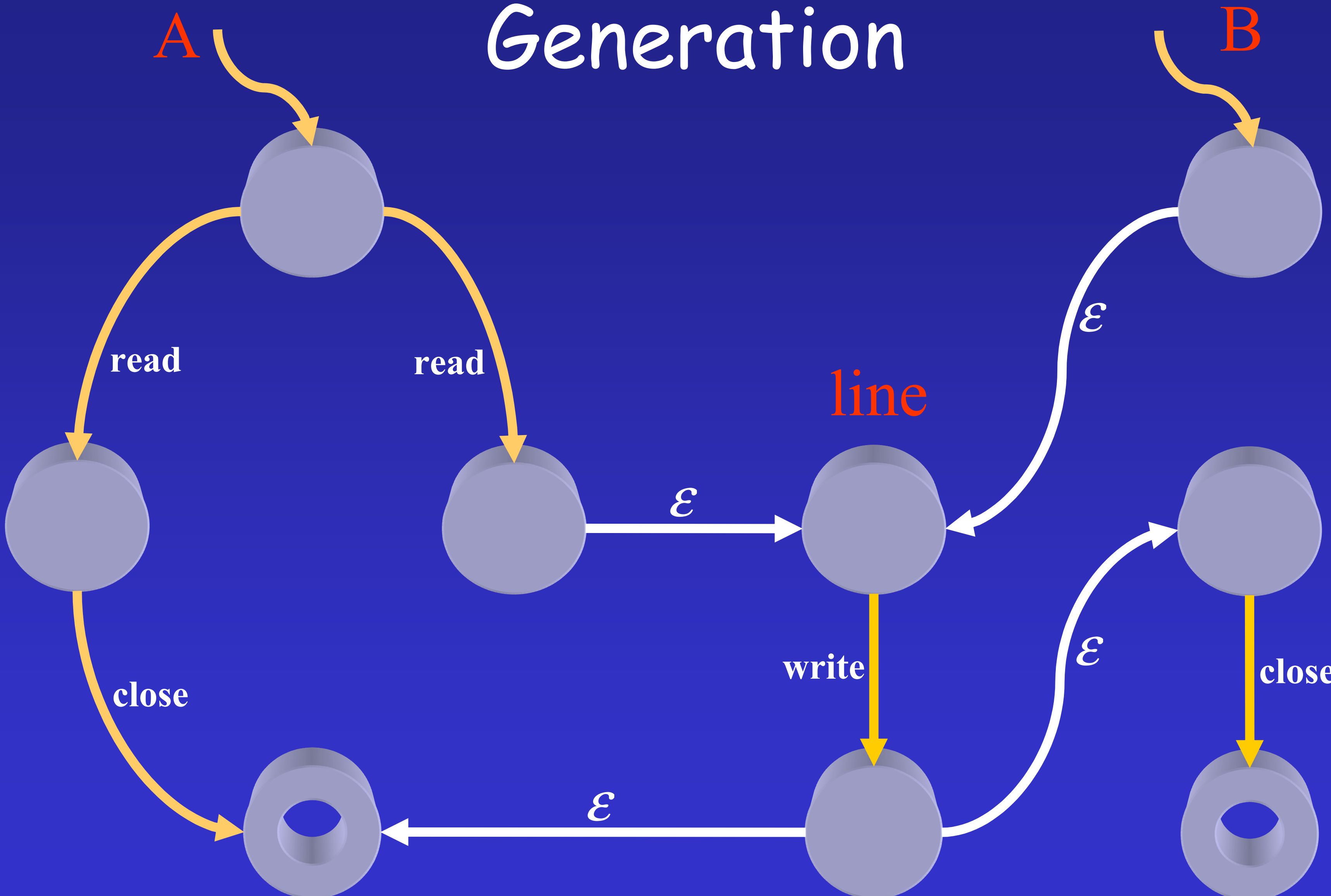
Interprocedural Model Generation



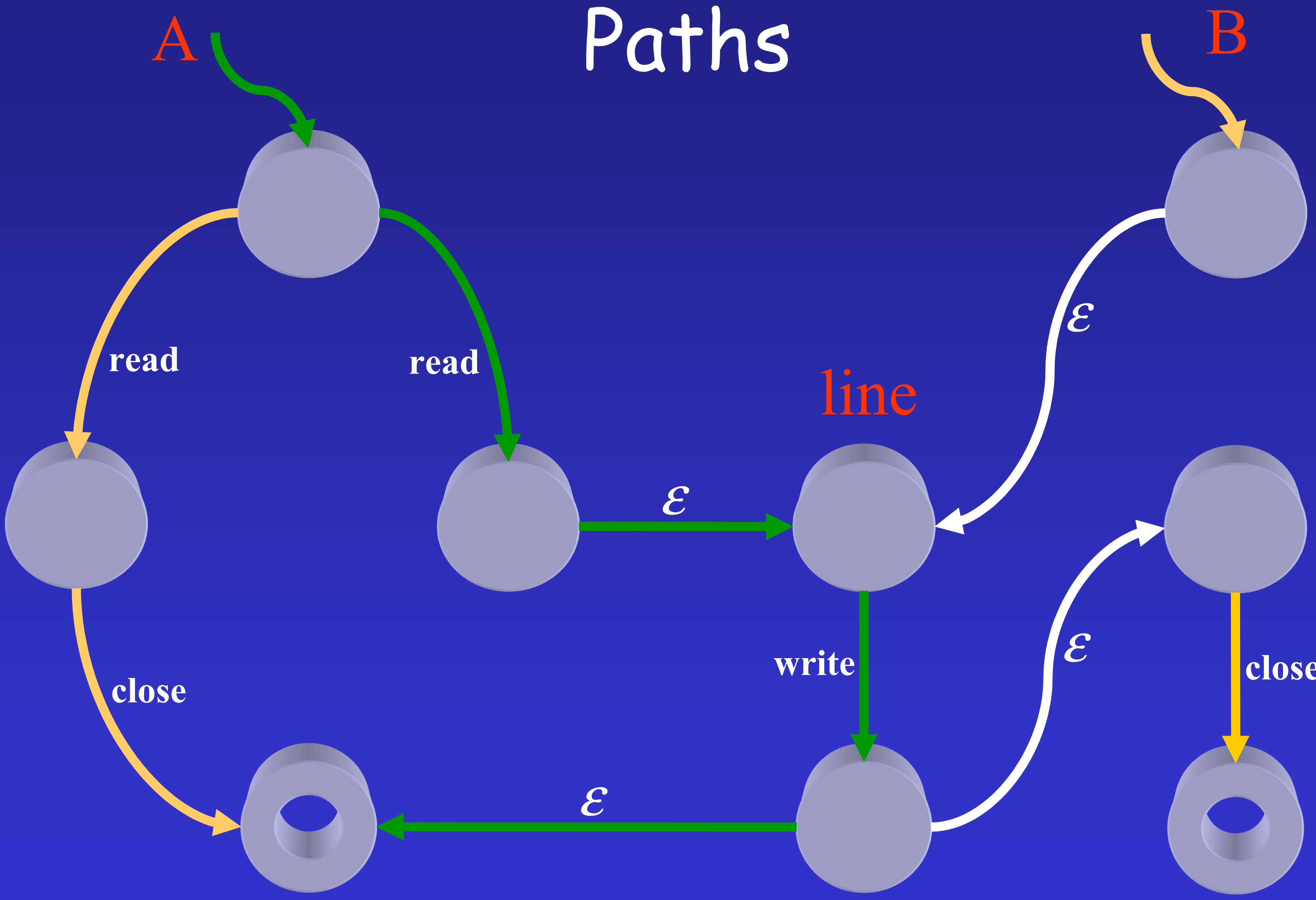
Interprocedural Model Generation



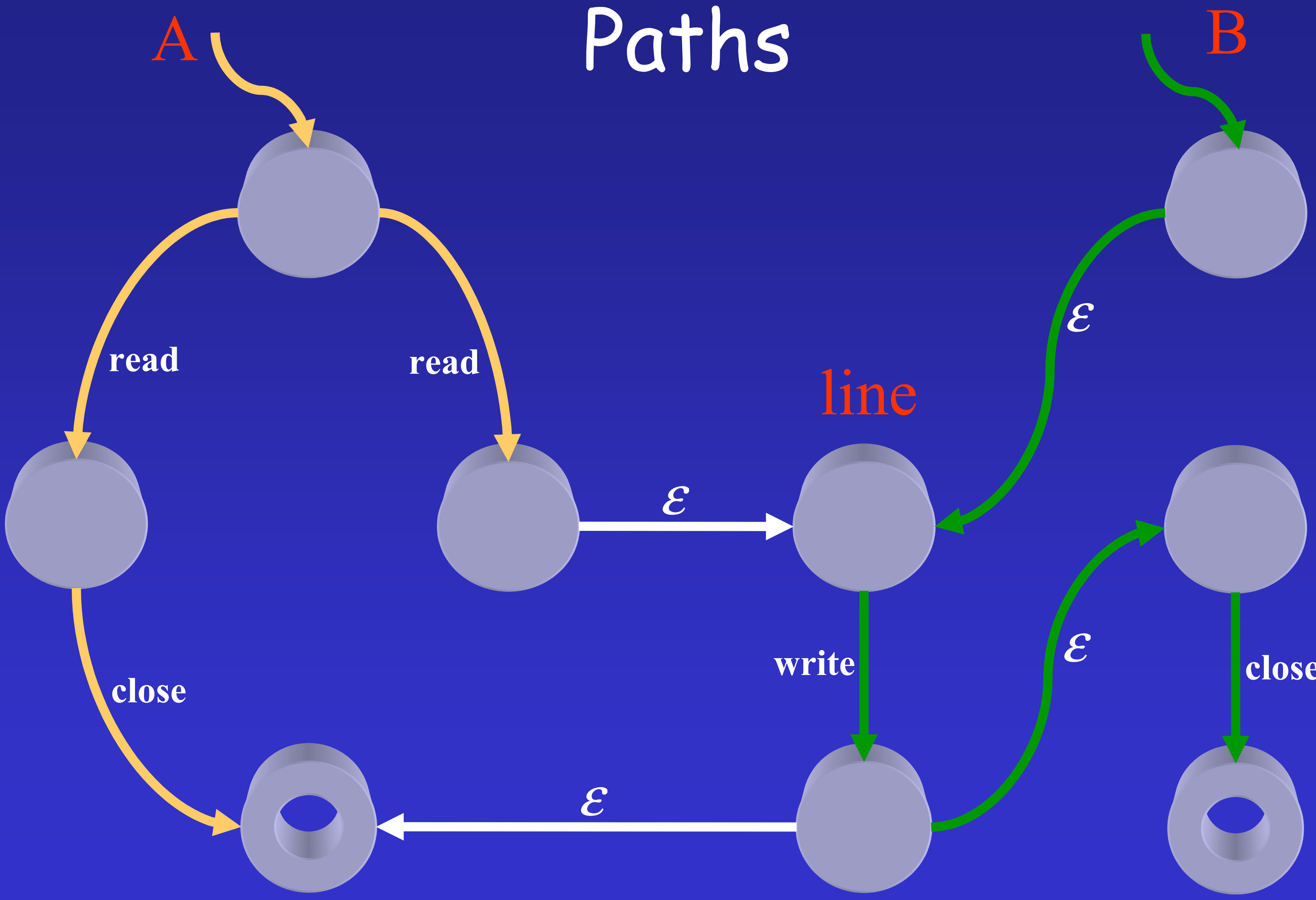
Interprocedural Model Generation



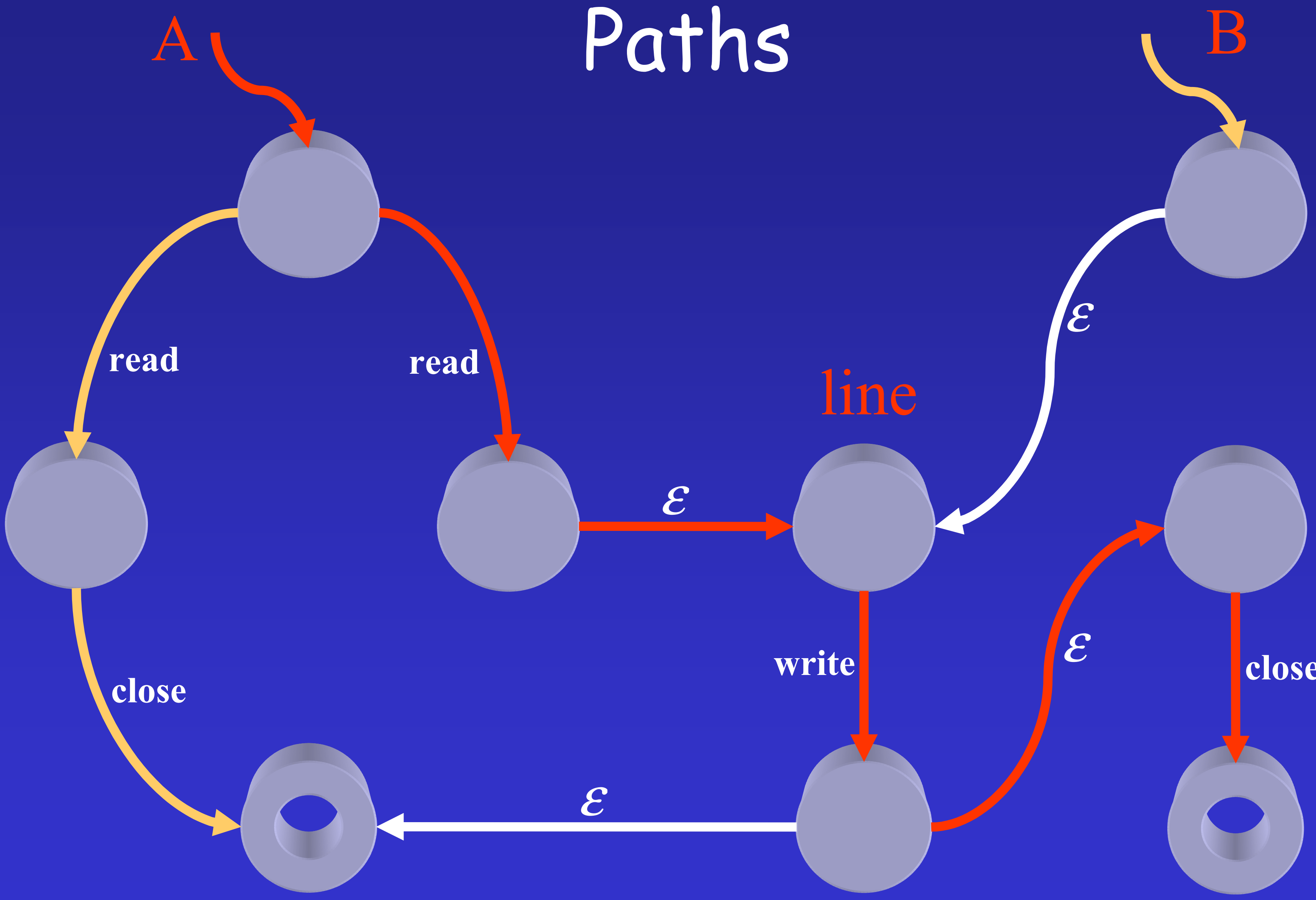
Possible Paths



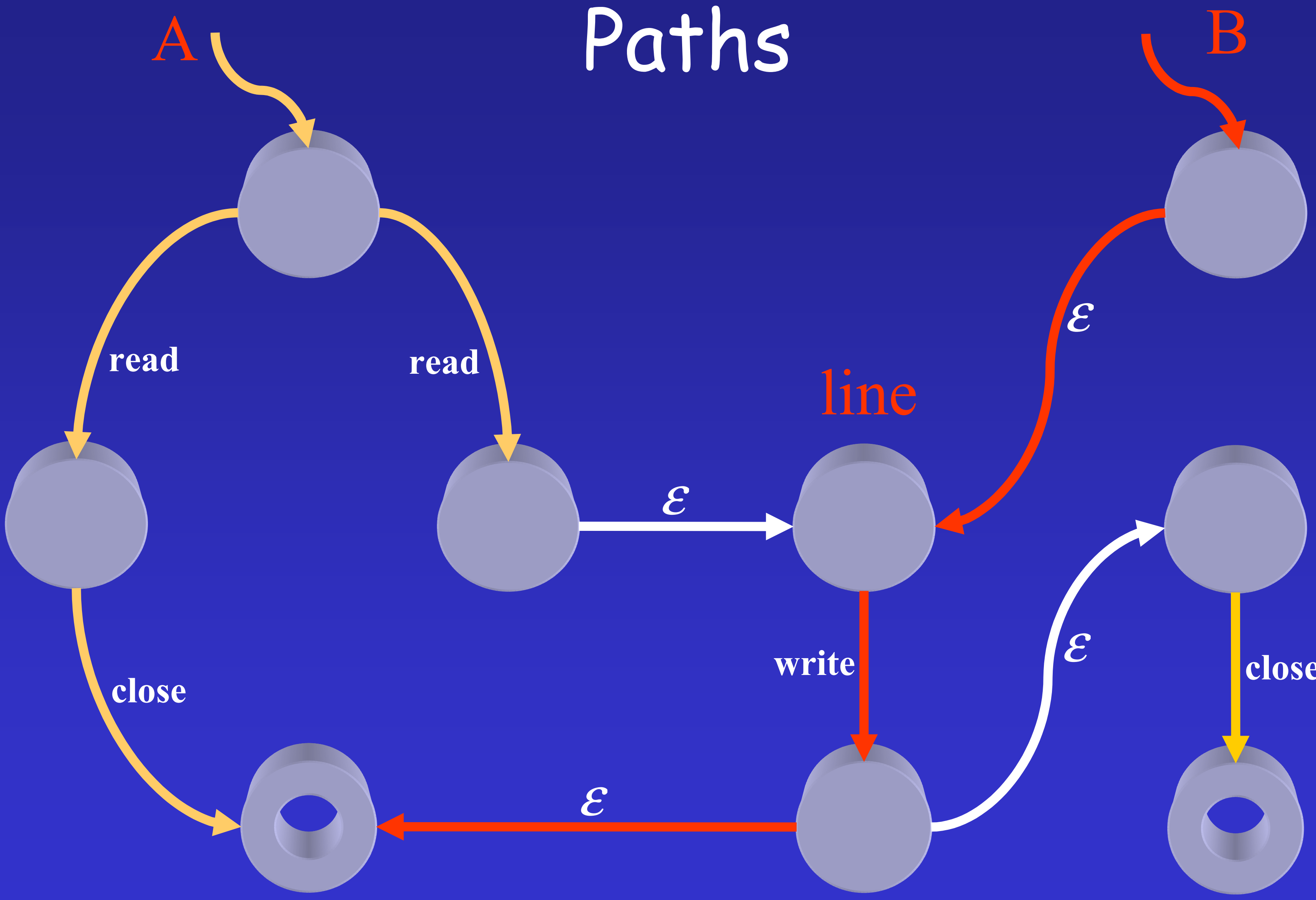
Possible Paths



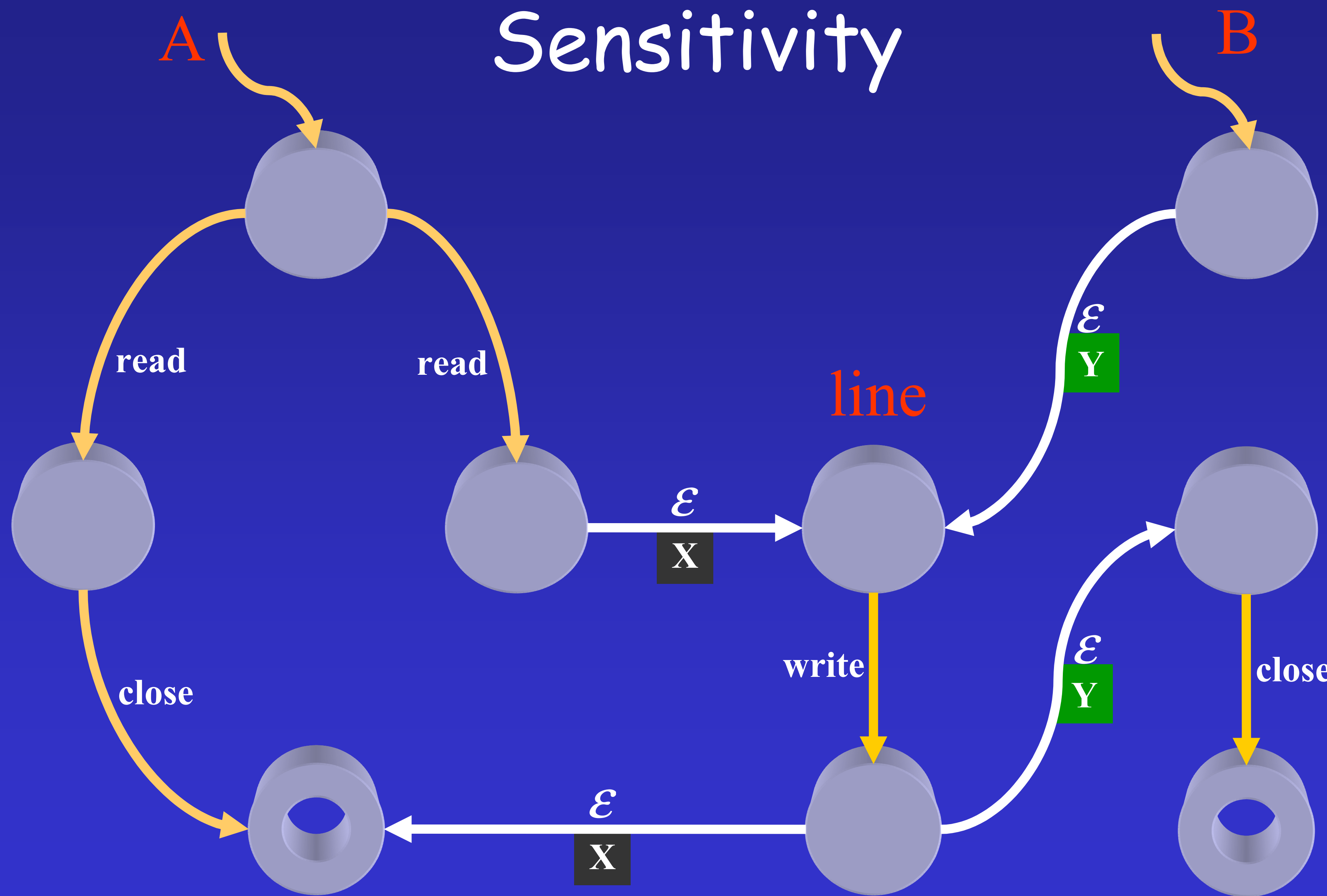
Impossible Paths



Impossible Paths

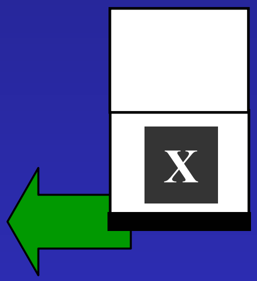


Adding Context Sensitivity



PDA State Explosion

- ε-edge identifiers maintained on a stack
  - Stack may grow to be unbounded



- Solution:
  - Bound the maximum size of the runtime stack
  - A regular language overapproximation of the context-free language of the PDA

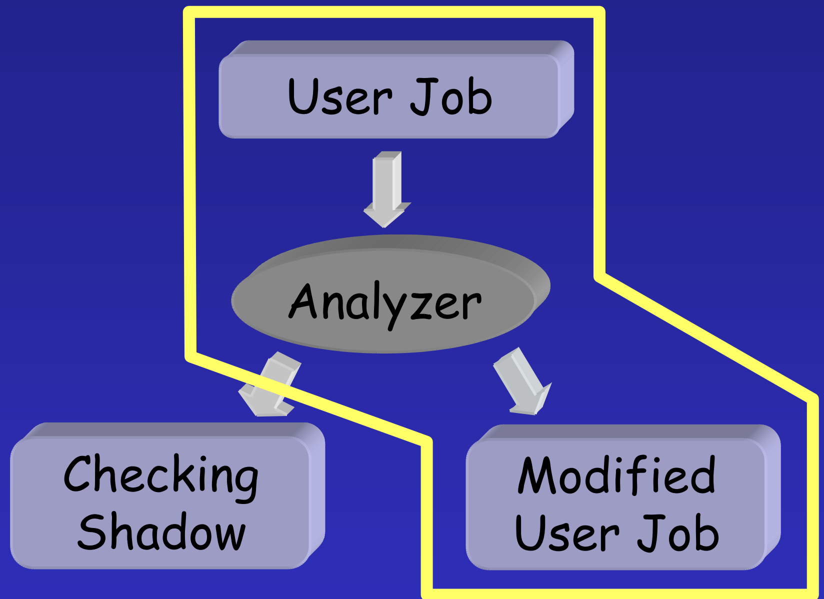
Data Flow Analysis

```
function:
  save %sp, 0x96, %sp
  cmp %i0, 0
  bge L1
  mov 15, %o1
  call read
  mov 0, %o0
  call line
  nop
  b L2
  nop
L1:
  call read
  mov %i0, %o0
  call close
  mov %i0, %o0
L2:
  ret
  restore
```

Argument recovery

- Statically known arguments constrain remote calls
- Reduces opportunity given to attackers

Rewriting User Job



Binary Program → Rewritten Binary

Call Site Renaming

```
function:
  save %sp, 0x96, %sp
  cmp $i0, 0
  bge L1
  mov 15, %o1
  call read
  mov 0, %o0
  call line
  nop
  b L2
  nop
L1:
  call read
  mov %i0, %o0
  call close
  mov %i0, %o0
L2:
  ret
  restore
```

- Give each monitored call site a unique name
- Associates arguments with call sites
- Obfuscation
- Reduces nondeterminism

Call Site Renaming

```
function:
  save %sp, 0x96, %sp
  cmp $i0, 0
  bge L1
  mov 15, %o1
  call _638
  mov 0, %o0
  call line
  nop
  b L2
  nop
L1:
  call read
  mov %i0, %o0
  call close
  mov %i0, %o0
L2:
  ret
  restore
```

- Give each monitored call site a unique name
- Associates arguments with call sites
- Obfuscation
- Reduces nondeterminism

Call Site Renaming

```
function:
  save %sp, 0x96, %sp
  cmp $i0, 0
  bge L1
  mov 15, %o1
  call _638
  mov 0, %o0
  call line
  nop
  b L2
  nop
L1:
  call _83
  mov %i0, %o0
  call close
  mov %i0, %o0
L2:
  ret
  restore
```

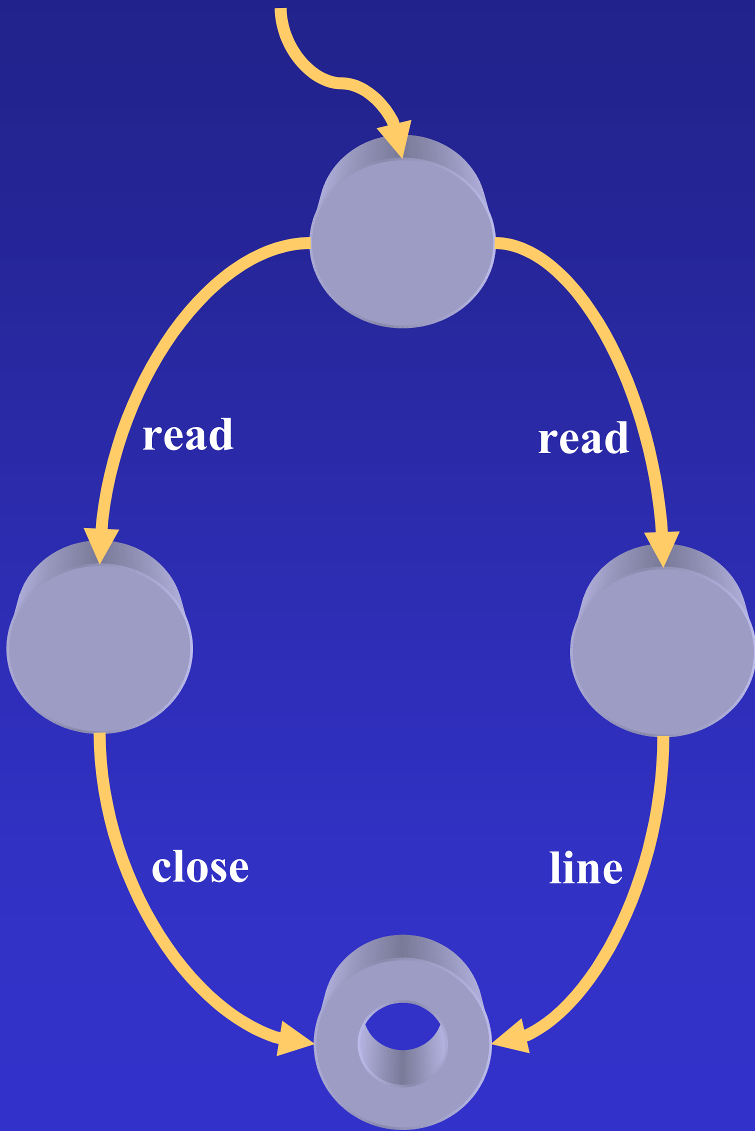
- Give each monitored call site a unique name
- Associates arguments with call sites
- Obfuscation
- Reduces nondeterminism

Call Site Renaming

```
function:
  save %sp, 0x96, %sp
  cmp $i0, 0
  bge L1
  mov 15, %o1
  call _638
  mov 0, %o0
  call line
  nop
  b L2
  nop
L1:
  call _83
  mov %i0, %o0
  call _1920
  mov %i0, %o0
L2:
  ret
  restore
```

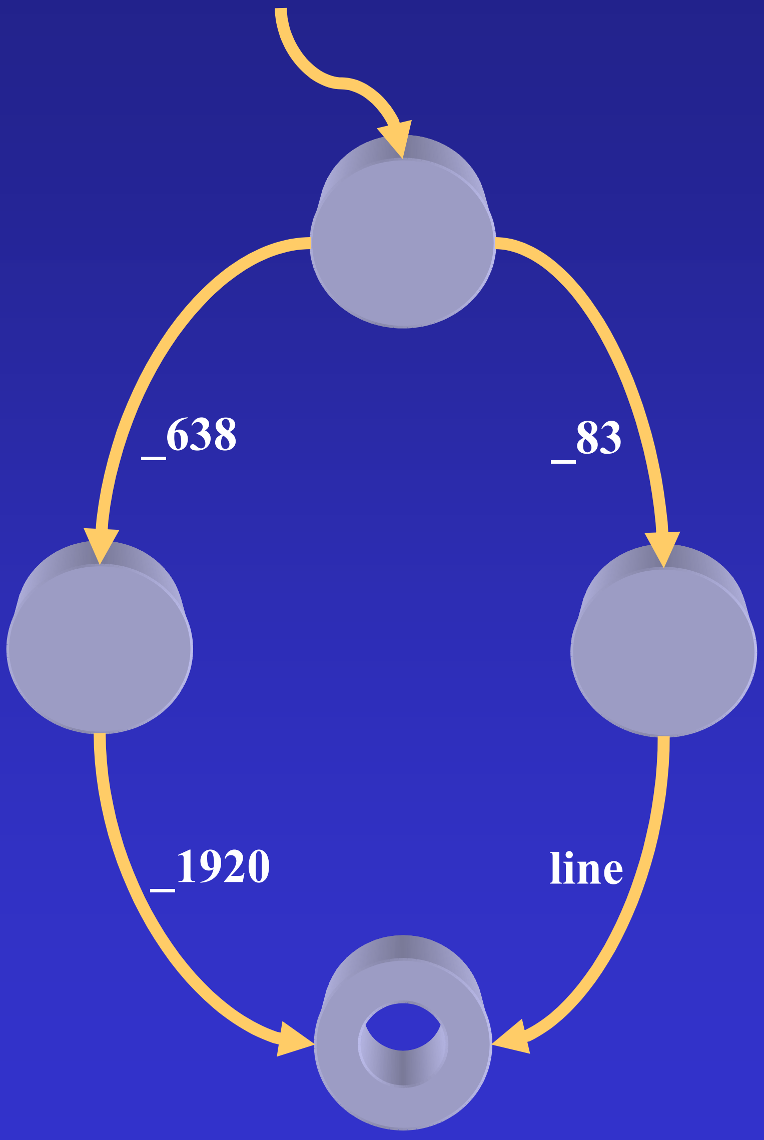
- Give each monitored call site a unique name
- Associates arguments with call sites
- Obfuscation
- Reduces nondeterminism

Call Site Renaming



- Give each monitored call site a unique name
- Associates arguments with call sites
- Obfuscation
- Reduces nondeterminism

Call Site Renaming



- Give each monitored call site a unique name
- Associates arguments with call sites
- Obfuscation
- Reduces nondeterminism

Prototype Implementation

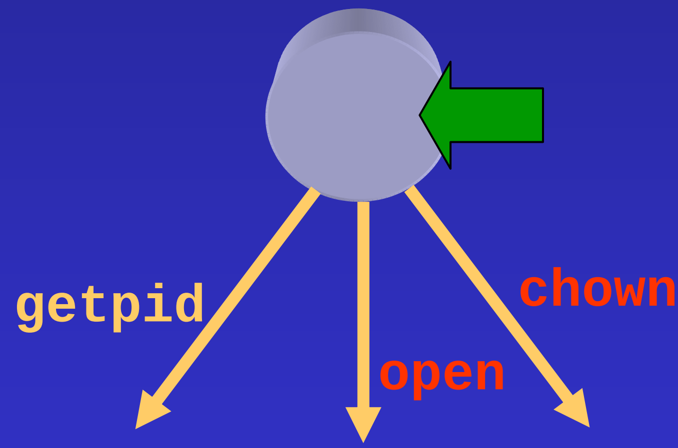
- Simulates remote execution environment
- Measure model precision
- Measure runtime overheads
- Measure the effect of changing maximum stack depth on bounded PDA model

Test Programs

	Program Size in Instructions	Workload
gzip	56,686	Compress a 13 MB file
GNU finger	95,534	Finger 3 non-local users
procmail	107,167	Process 1 incoming email message

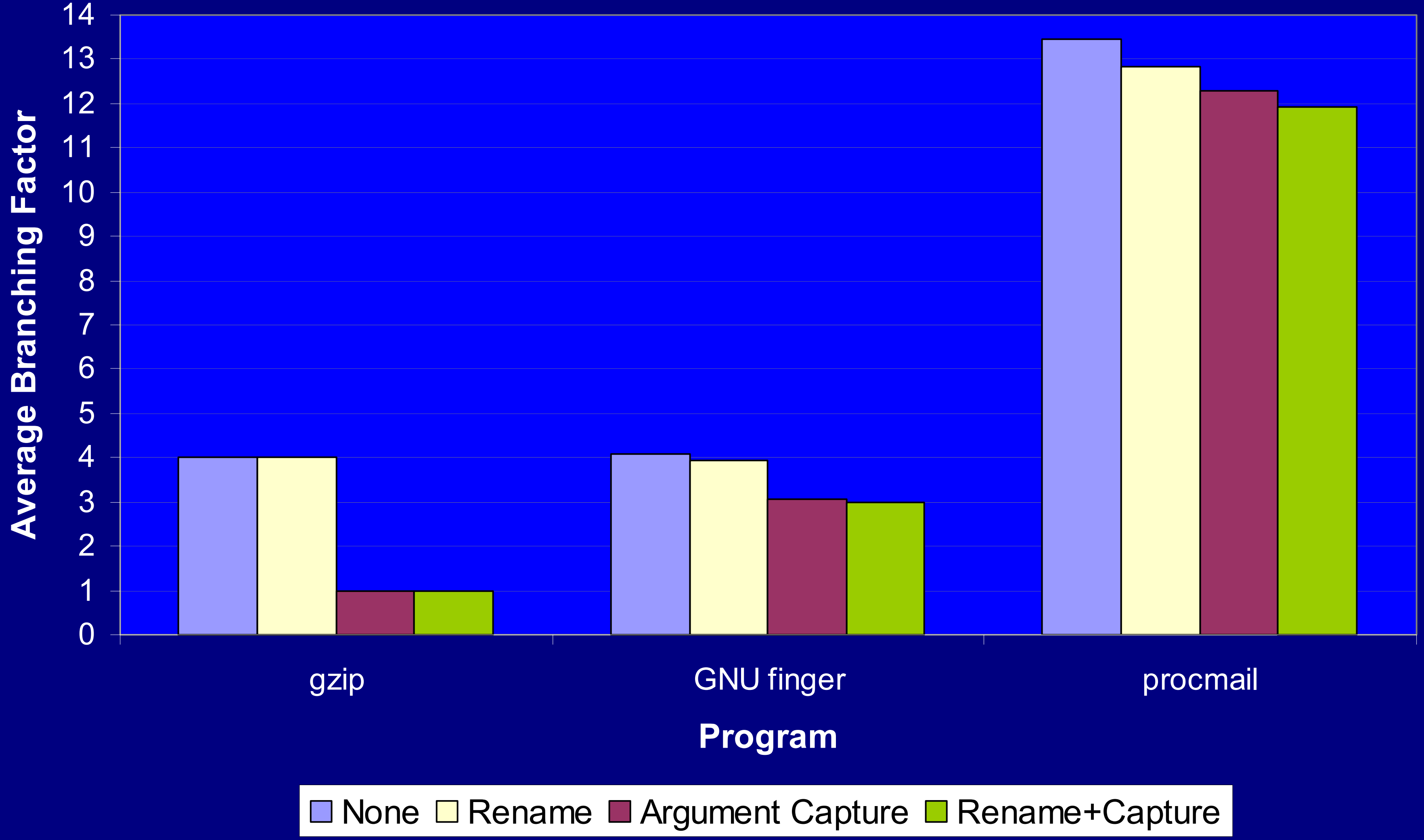
Precision Metric

- Average branching factor

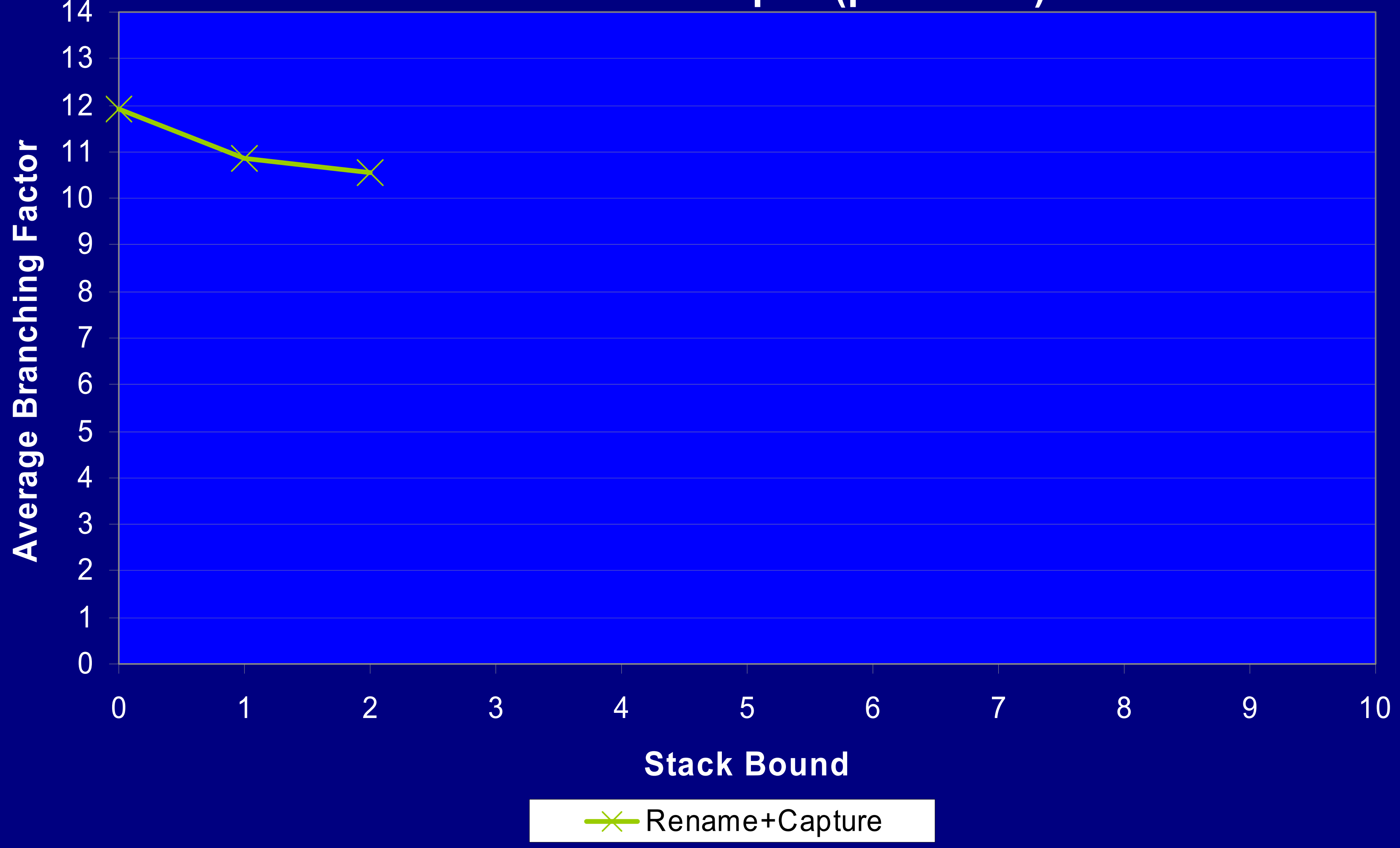


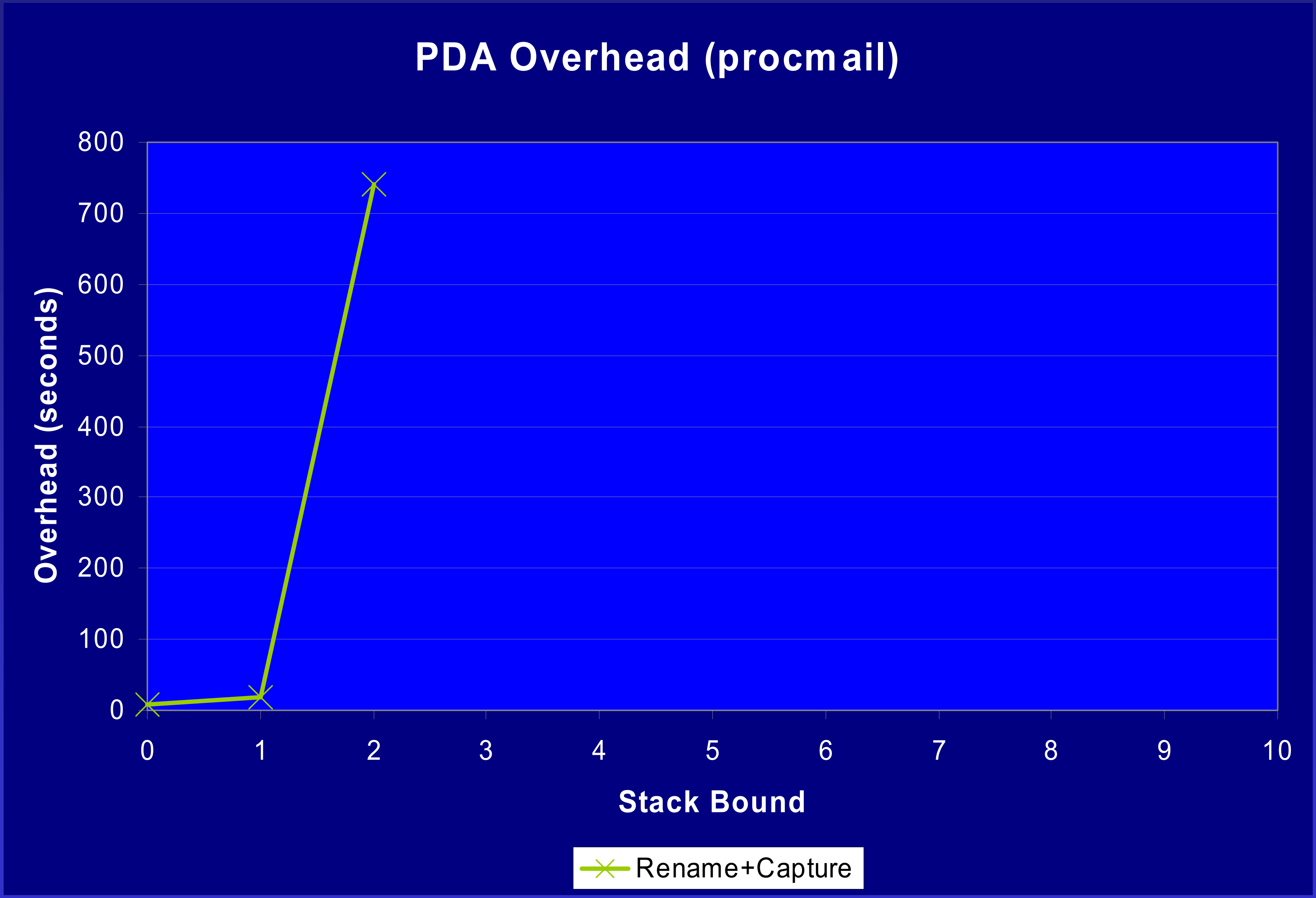
- Lower values indicate greater precision

Optimizations Improve Precision



PDA Precision Improves with Increased Stack Depth (procmail)




PDA Overhead (procmail)
Overhead (seconds)
800
700
600
500
400
300
200
100
0
0
1
2
3
4
5
6
7
8
9
10
Stack Bound
Rename+Capture

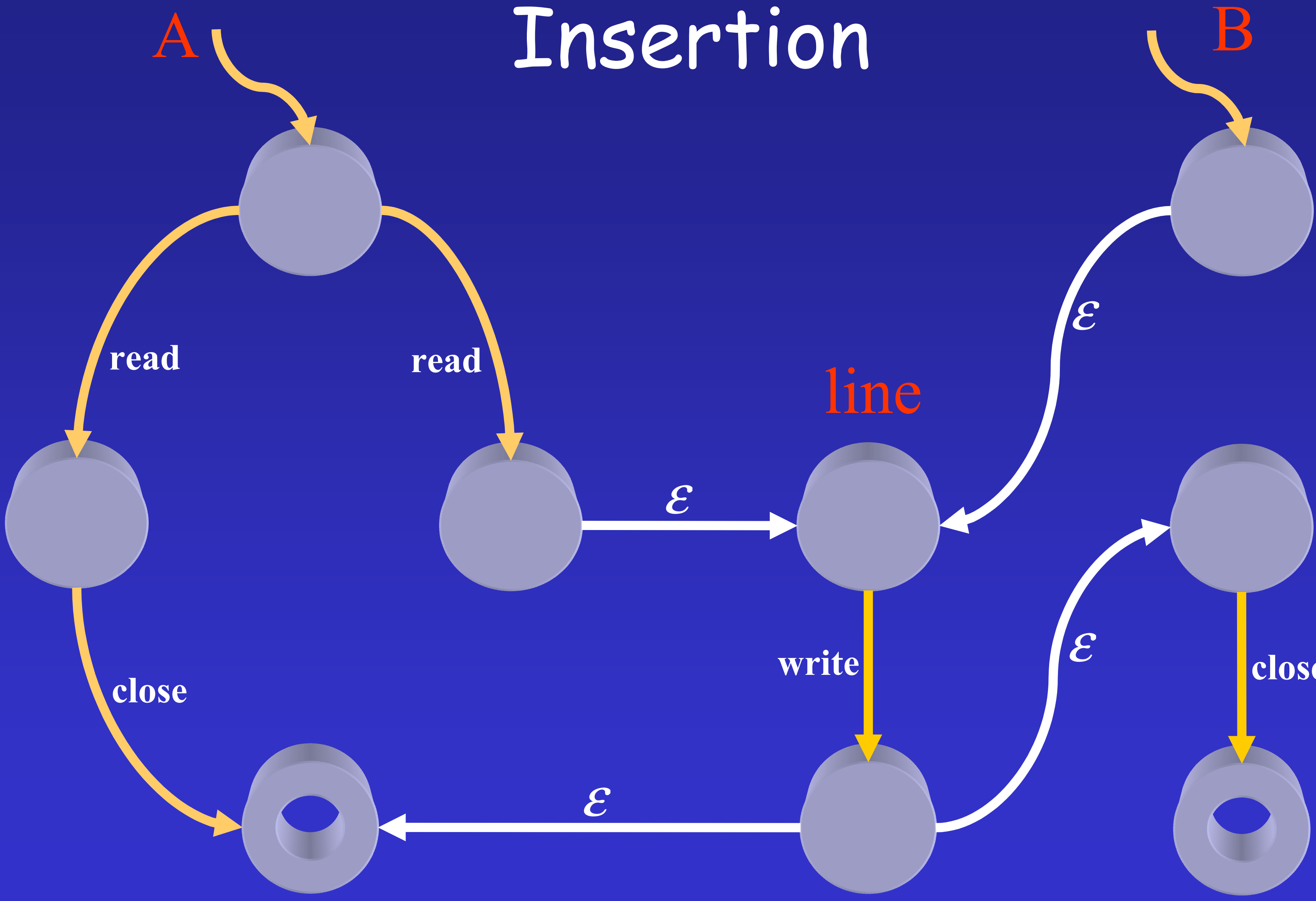
Recent Work

- Improving precision with null calls
  - Surprise! PDA performance improves
- Analysis of shared objects

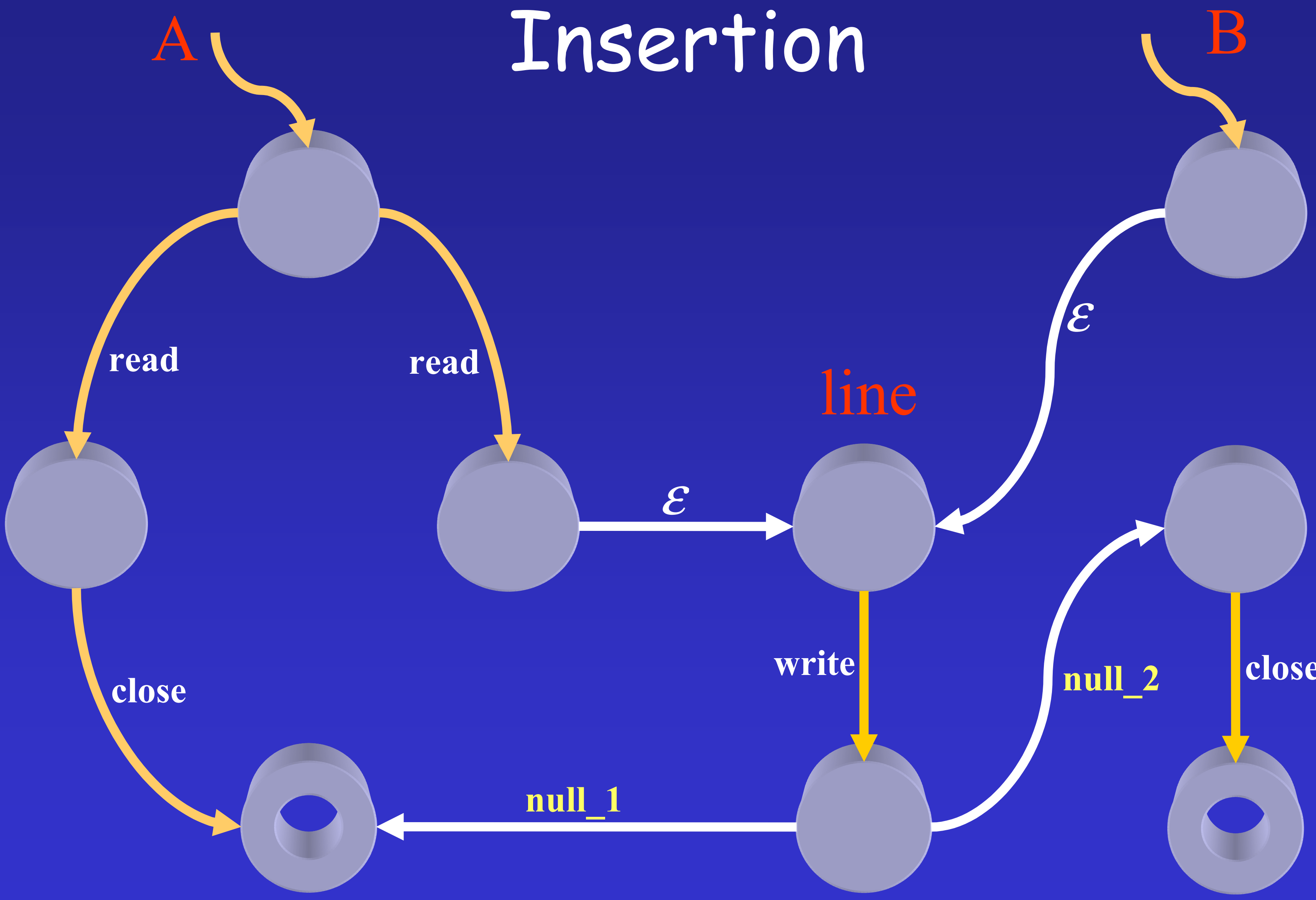
Null Calls

- Observation: PDA is more precise than NFA because it provides context sensitivity
- Idea: Insert null calls into NFA model to add some context sensitivity without suffering runtime cost of PDA

Null Call Insertion



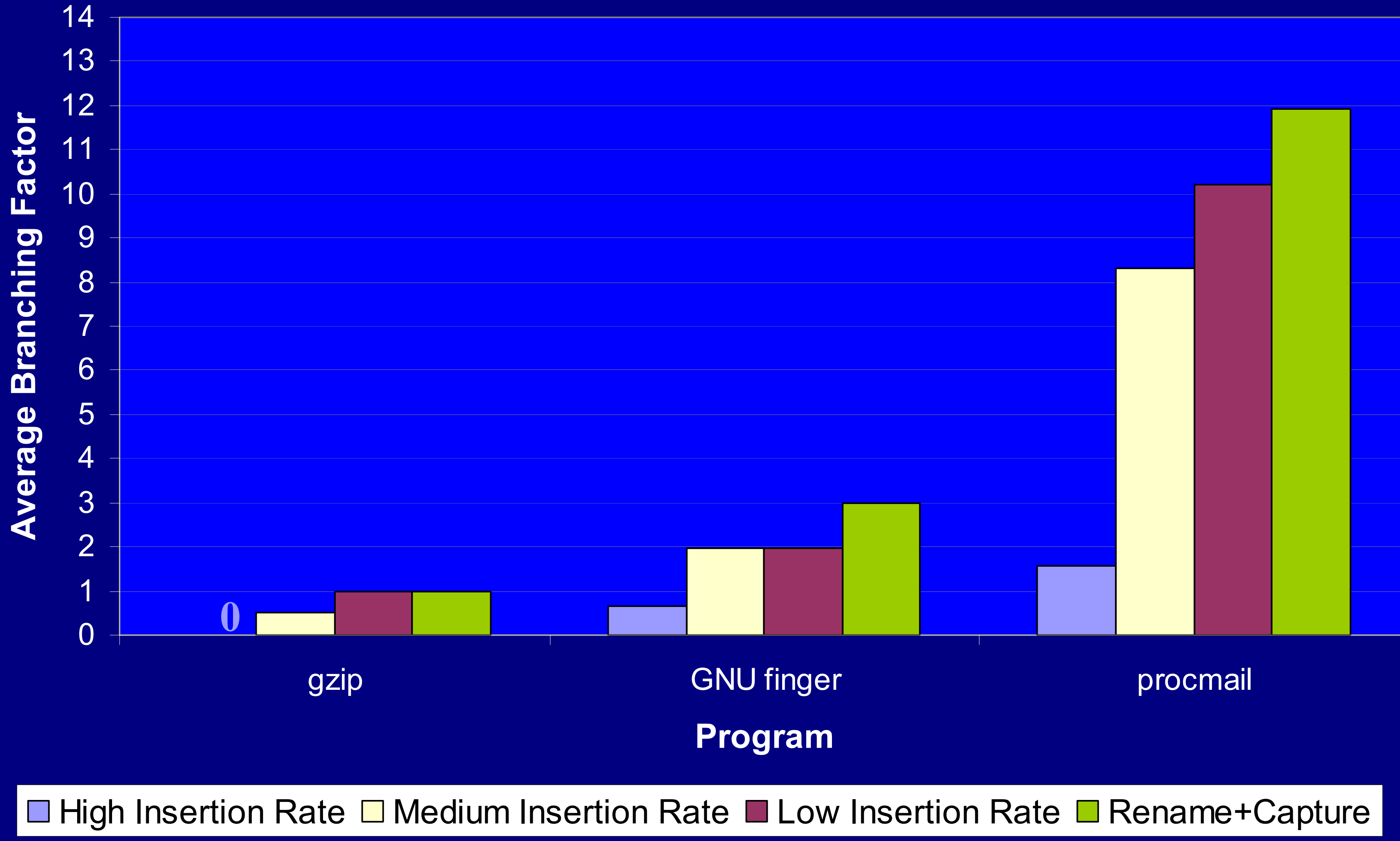
Null Call Insertion



Null Call Experiments

- Inserted null calls at 3 rates
  - High: At entries of functions with fan-in of 2 or greater
  - Medium: At entries of functions with fan-in of 5 or greater
  - Low: At entries of functions with fan-in of 10 or greater

Precision Improves with Null Calls

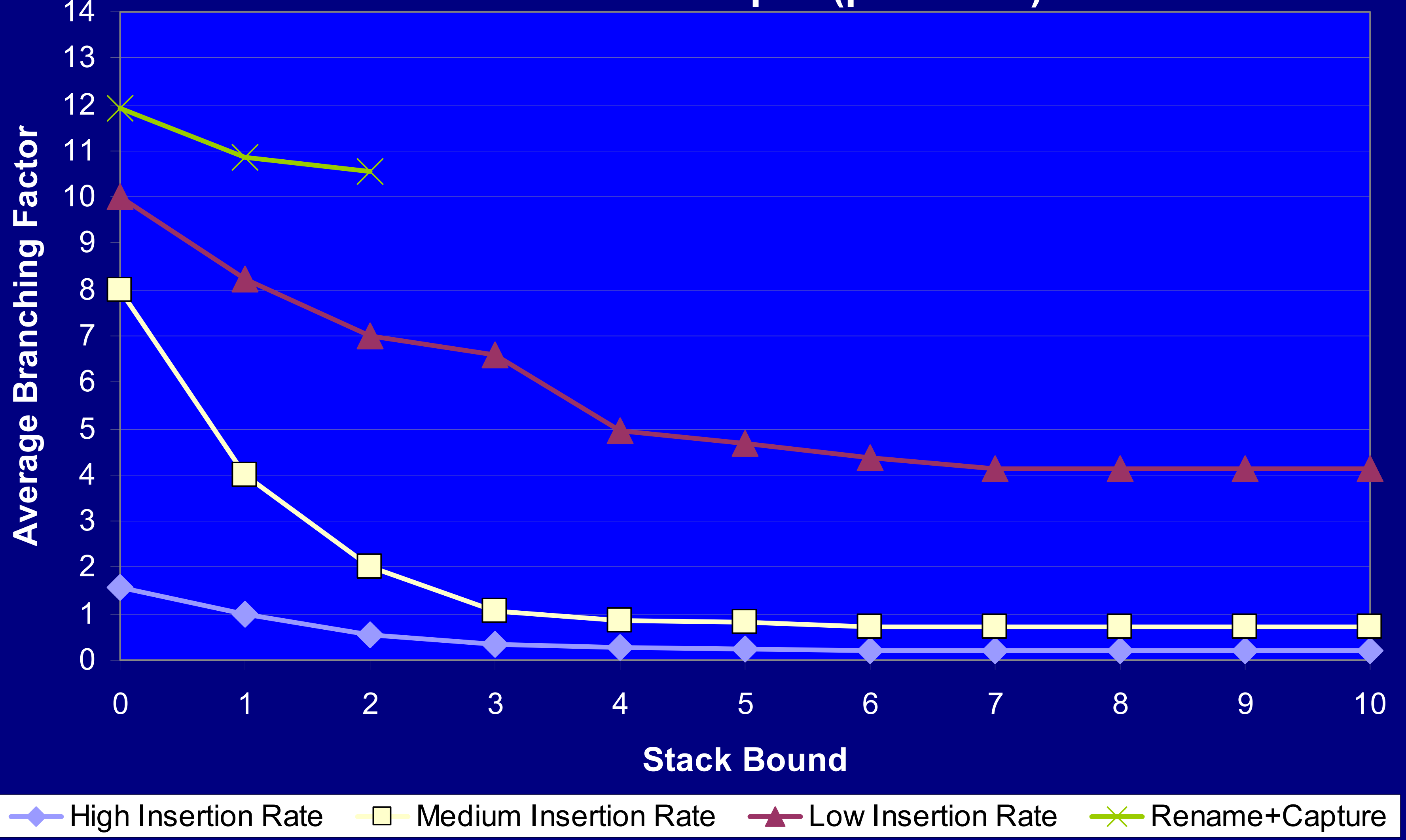


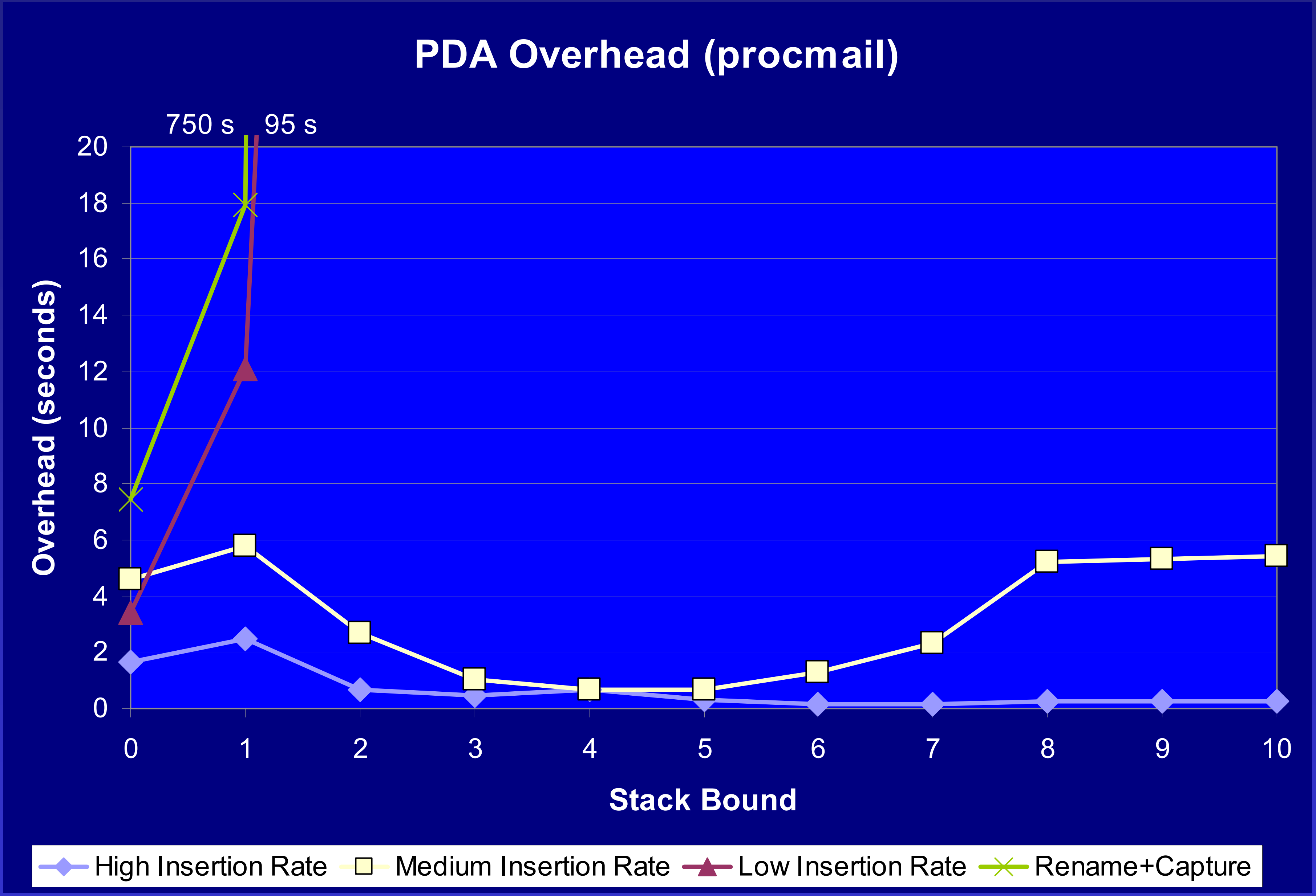
Null Call Costs: Monitoring Overhead & Bandwidth

Insertion Rate	High	Medium	Low
gzip	747.0 %	< 0.1 %	< 0.1 %
GNU finger	0.1 %	0.1 %	< 0.1 %
procmail	0.8 %	1.1 %	0.7 %

gzip	4350.0 Kbps	5.6 Kbps	0.0 Kbps
GNU finger	14.1 Kbps	9.1 Kbps	0.9 Kbps
procmail	18.2 Kbps	13.1 Kbps	4.0 Kbps

PDA Precision Improves With Null Call Insertion & Increased Stack Depth (procmail)




PDA Overhead (procmail)
750 s
95 s
Overhead (seconds)
20
18
16
14
12
10
8
6
4
2
0
0
1
2
3
4
5
6
7
8
9
10
Stack Bound
High Insertion Rate
Medium Insertion Rate
Low Insertion Rate
Rename+Capture

Analyzing Shared Object Code

- Two new difficulties
  - Relocatable object code
  - Interprocedural data flows

Relocatable Object Code

- Data tables filled out dynamically at load time
- Data table recovery
  - Recover relocation tables
  - Simulate action of run-time linker to resolve table values
- Enables improved analysis
  - Trace global data accesses
  - Follow jumps through table values

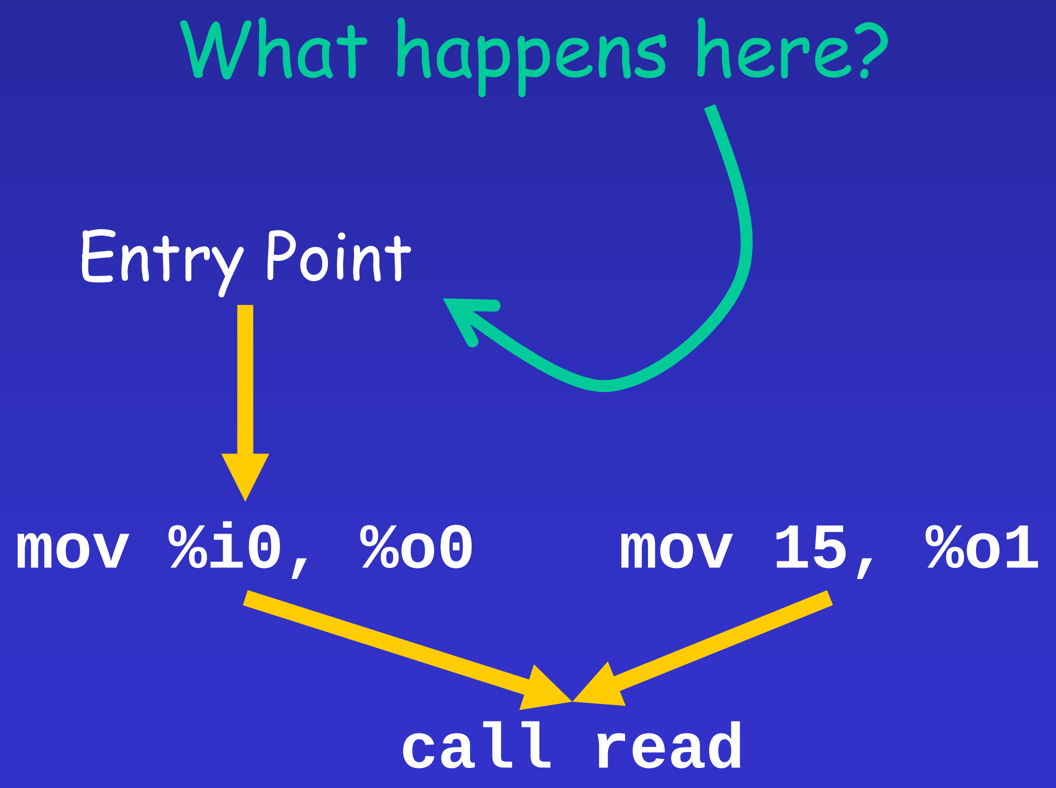
Data Flow Analysis

Argument recovery technique

- Slice on each register of interest to build a data dependence graph for the value
- Simulate the execution of the instructions in the dependence graph to reach final value

Argument Recovery

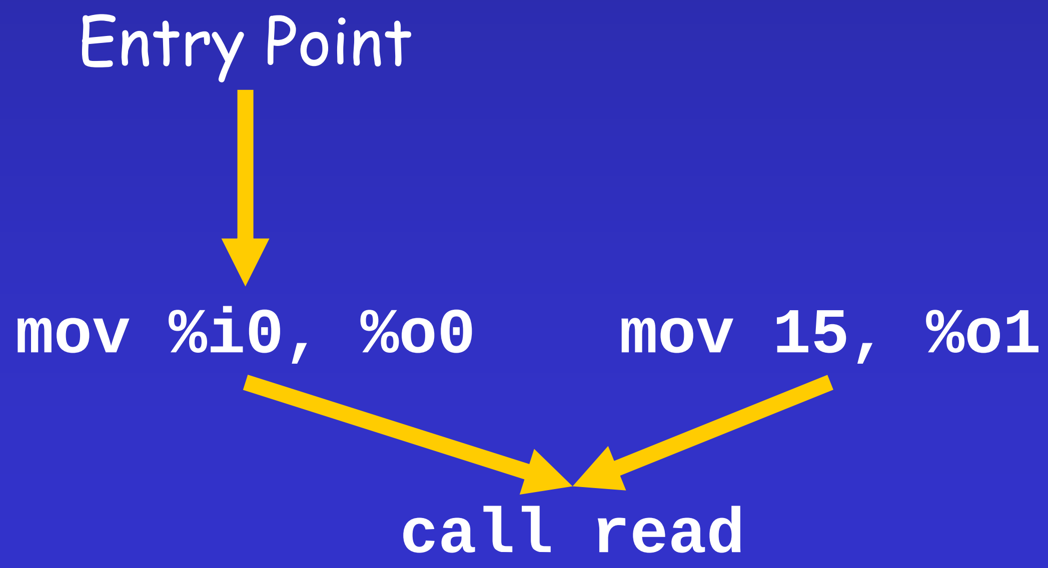
```
function:
   save %sp, 0x96, %sp
   cmp %i0, 0
   bge L1
   mov 15, %o1
   call read
   mov 0, %o0
   call line
   nop
   b L2
   nop
L1:
   call read
   mov %i0, %o0
   call close
   mov %i0, %o0
L2:
   ret
   restore
```



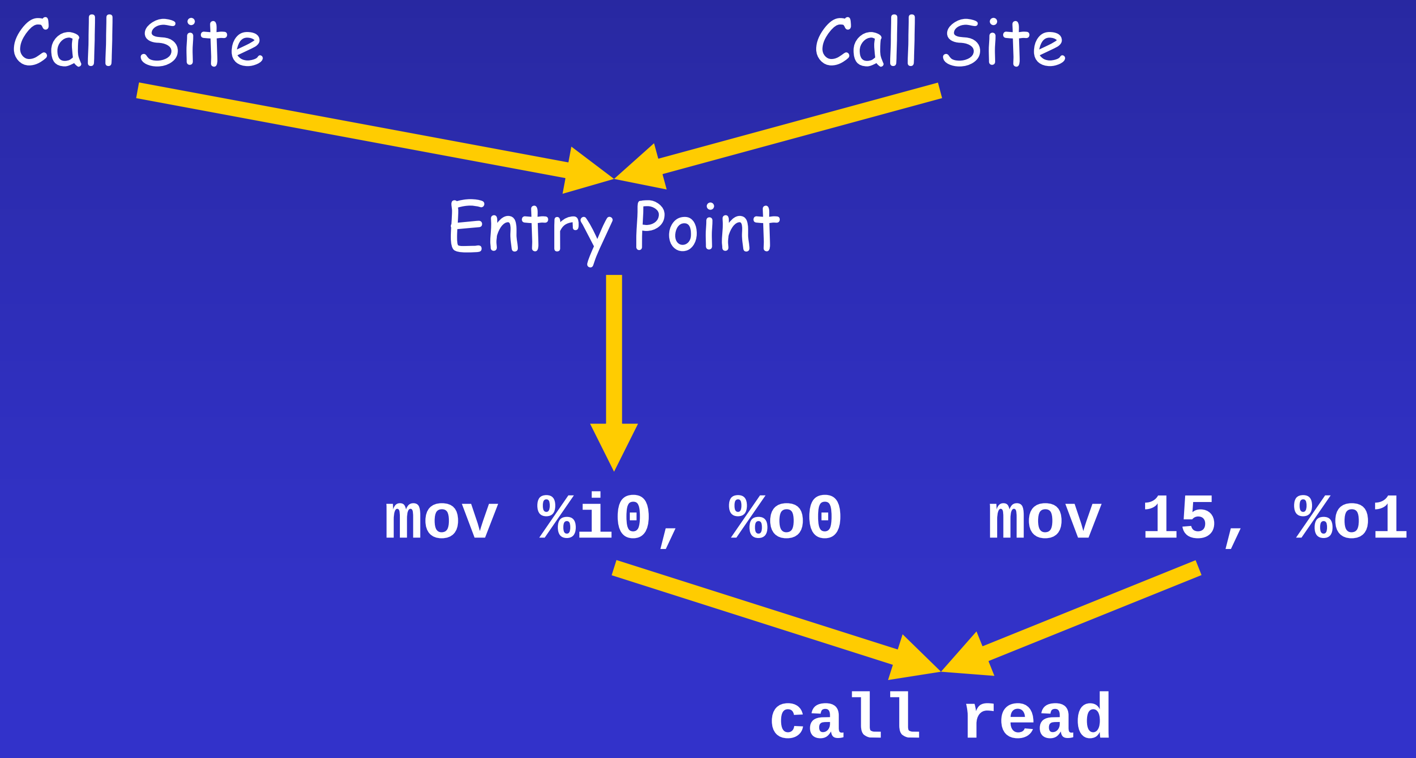
Argument Recovery

Interprocedural Slicing

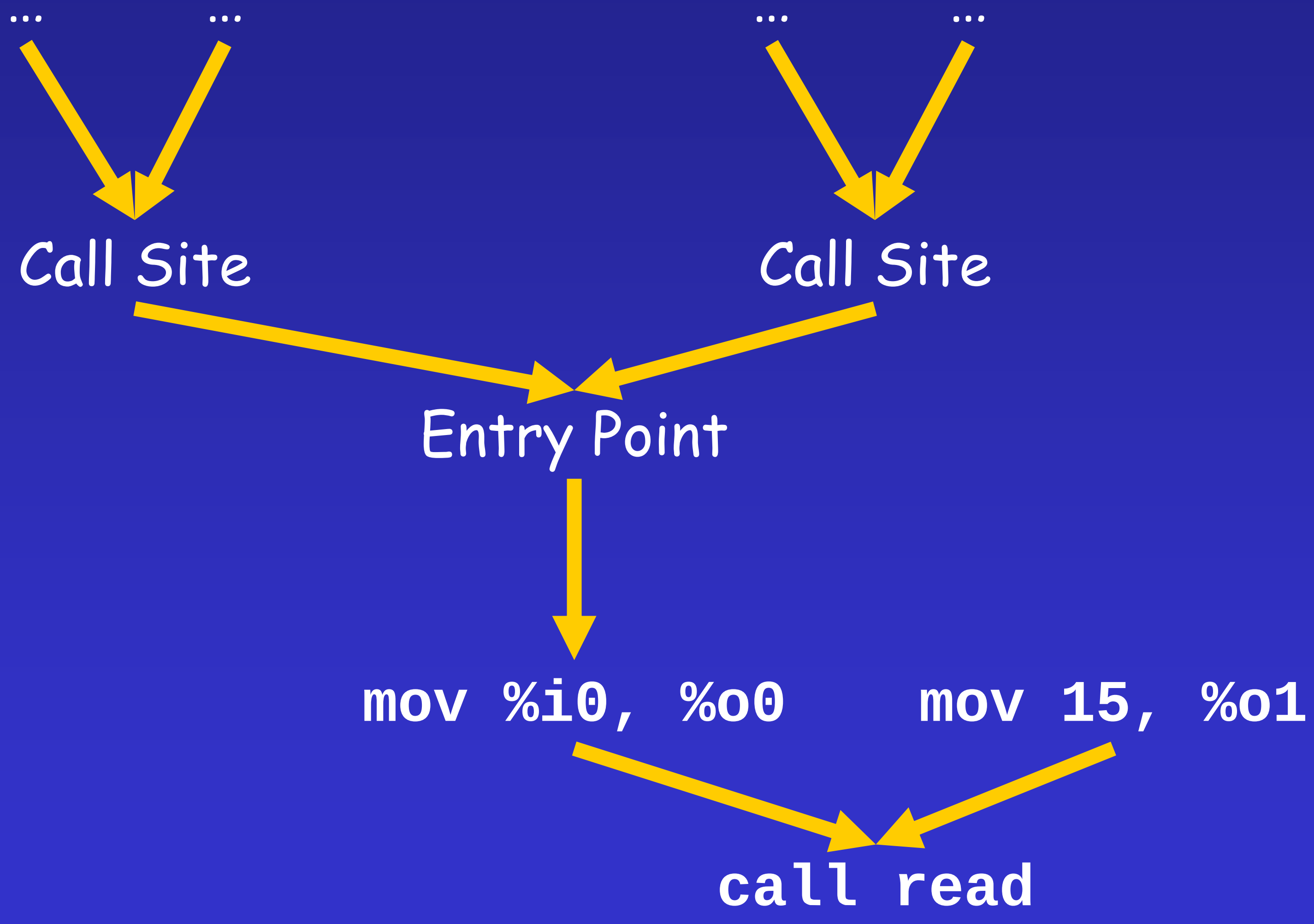
- Continue slice in calling functions



Argument Recovery



Argument Recovery



Argument Recovery

- Interprocedural slicing improves argument recovery
  - Imposes greater constraints upon attacker
- In shared objects, we can recover function pointers passed to library calls
  - Improves model precision

Analyzing Shared Object Code

Infrastructure Changes

- Both relocation table analysis & interprocedural slicing required modification of the analysis infrastructure

Status

- Recovering relocation tables is complete
- Interprocedural slicing is underway

Important Ideas

- Our work is specification-based monitoring with specifications generated automatically from binary code analysis.
- We enforce the specification by operating a finite state machine modeling correct execution.
- Null calls improve precision & PDA performance.
- Shared object analysis required addition of capabilities to the infrastructure.

Technical Agenda

- Integrating other specification sources
- Optimal null call insertion
- C++ vtable analysis

Specification-Based Analysis and Enforcement

Jonathon Giffin, Somesh Jha, Barton Miller
University of Wisconsin