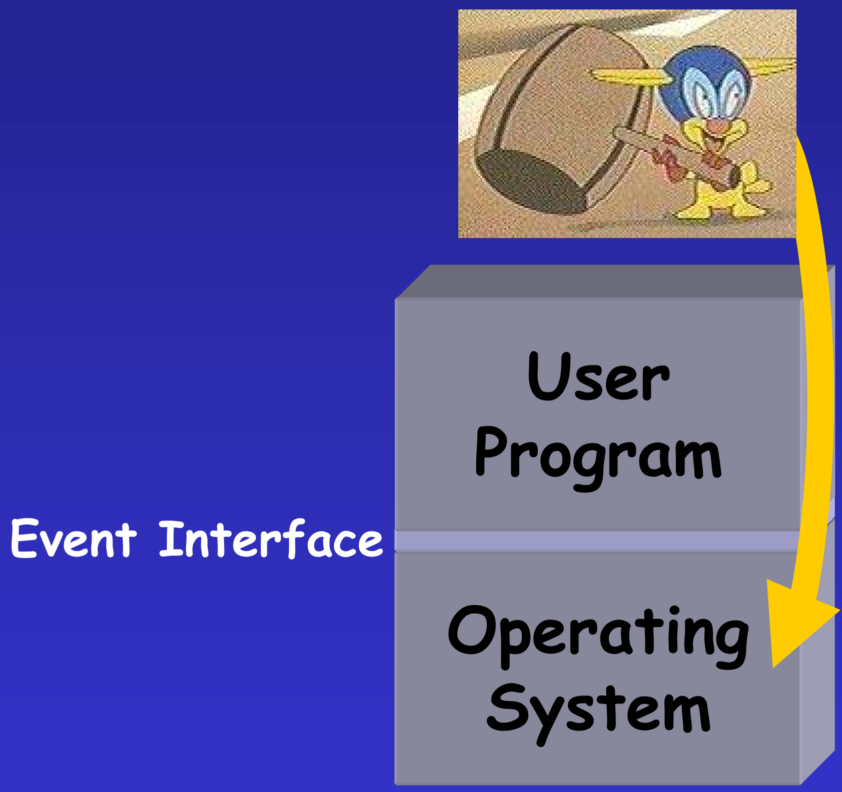
Specification-Based Monitoring: Improving Model Precision

Jonathon Giffin, Somesh Jha, Barton Miller
University of Wisconsin
Wisconsin Safety Analyzer

Overview

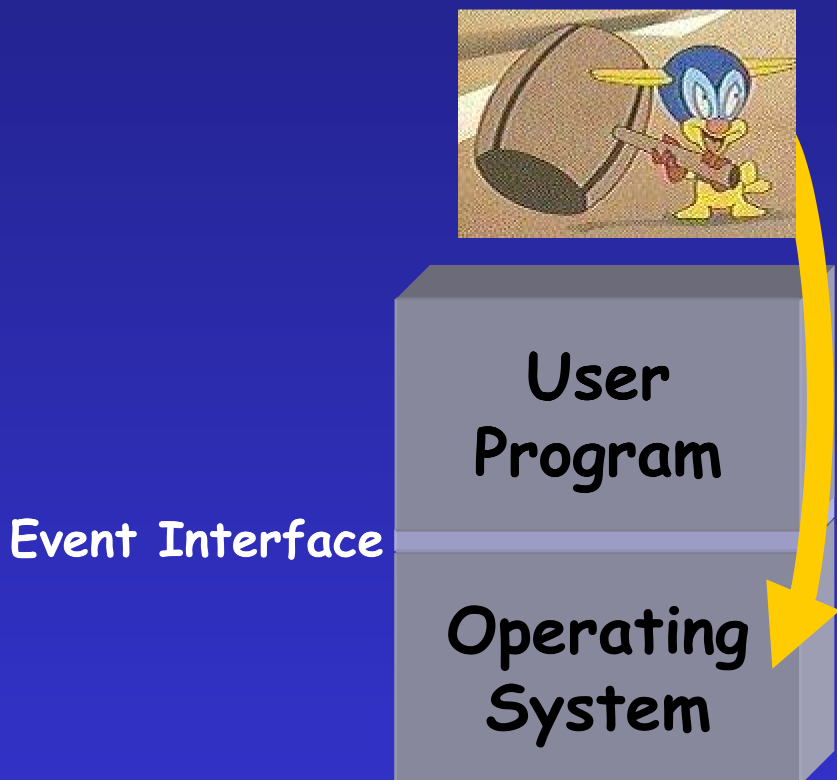
- The need for specification-based monitoring
- Static binary analysis for intrusion detection
- Strengthening our analyses:
  - Interprocedural data flow analysis
  - General argument representation
  - Intelligent null call insertion
- Performance results

Worldview



- User desires to run program
- Running program makes operating system requests
- Attacker uses running program to generate malicious requests

Worldview



- Attack goal:
  be creative…
  - Destruction
  - Information leaks
  - Service disruption

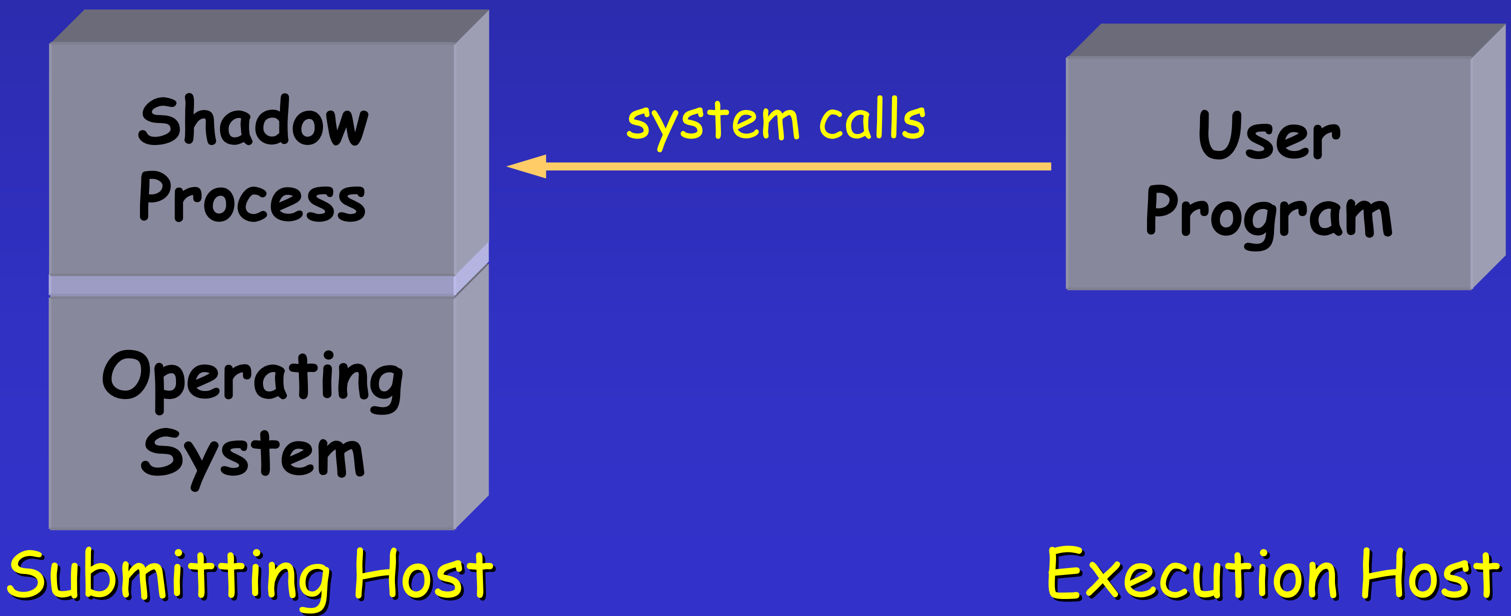
- Attack technique:
  run arbitrary code in the user program
  - Buffer overrun
  - Virus or worm
  - Condor lurking jobs

Example: SQL Slammer

- Worm activated Saturday morning
  - Caused worldwide service disruption
- Propagation: exploited buffer overrun in Microsoft SQL Server to execute arbitrary code
- Detection: SQL Server makes unexpected system calls
  - Arbitrary code differs from SQL code

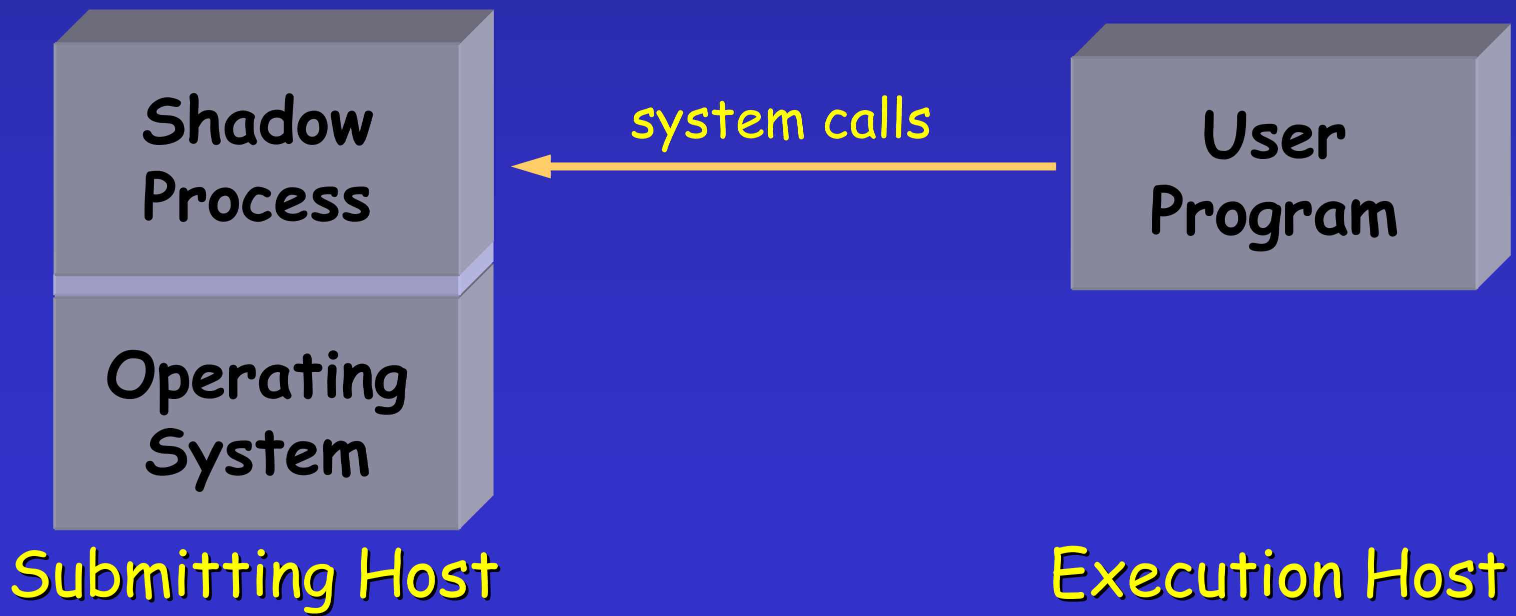
Example: The Condor Attack

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



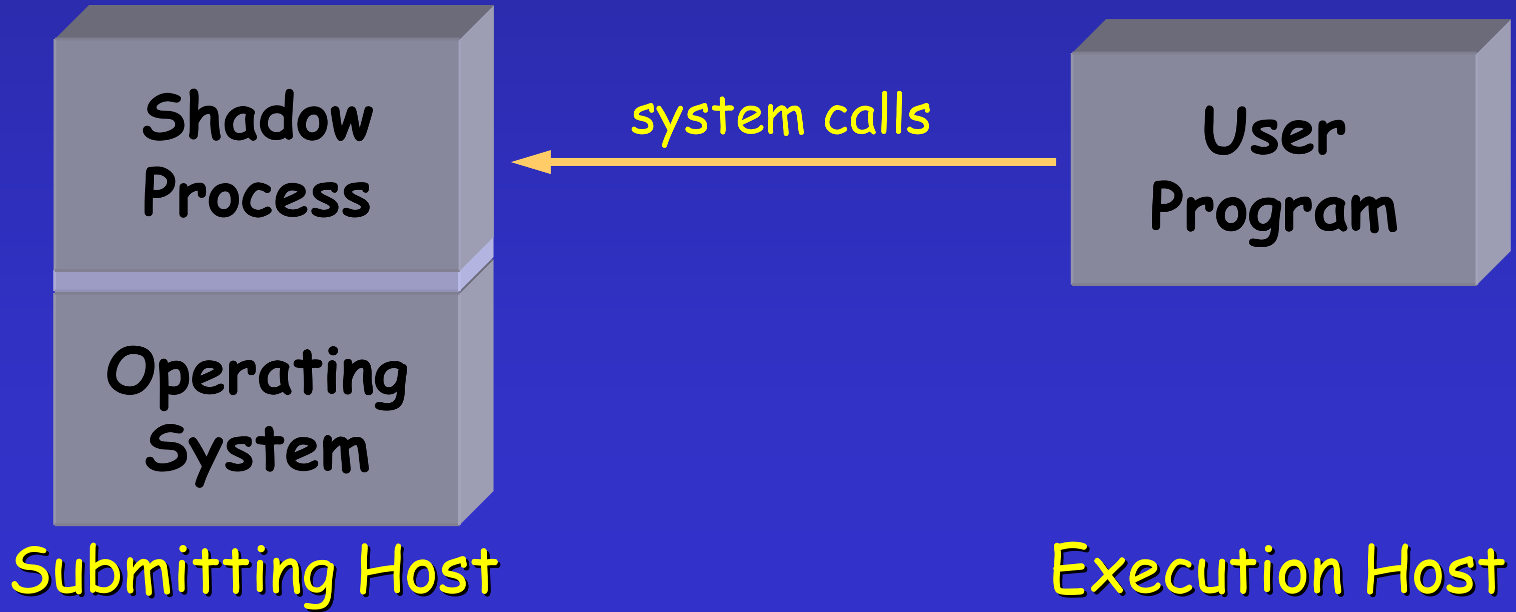
Example: The Condor Attack

- Attackers can manipulate remotely executing program
- Insert arbitrary code that takes control of link to user's machine

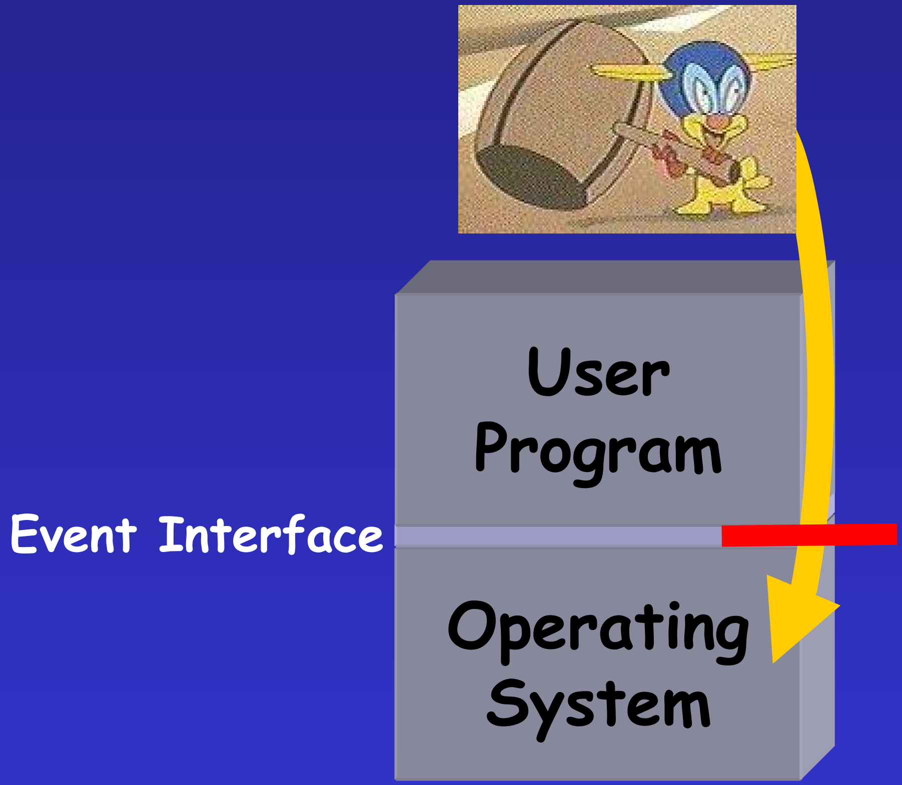


Example: The Condor Attack

- Detection: Remote user job makes unexpected remote system calls
  - Arbitrary code differs from job code

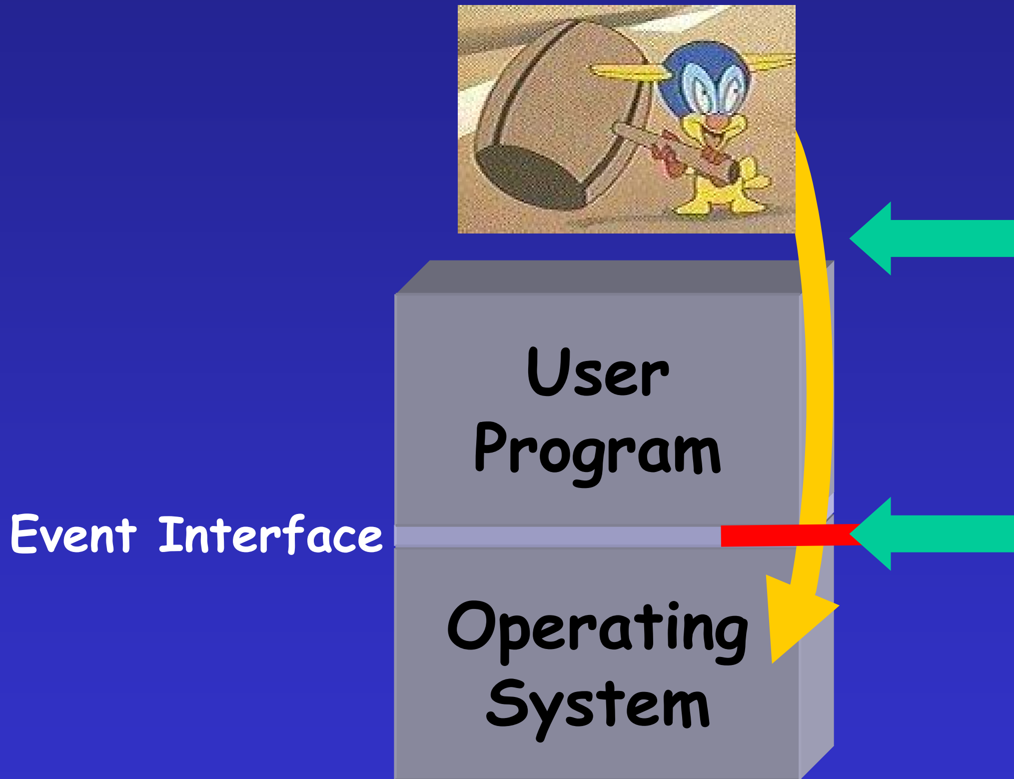


Our Objective



- Detect malicious activity before harm caused to local machine
- … before operating system executes malicious system call

Our Objective

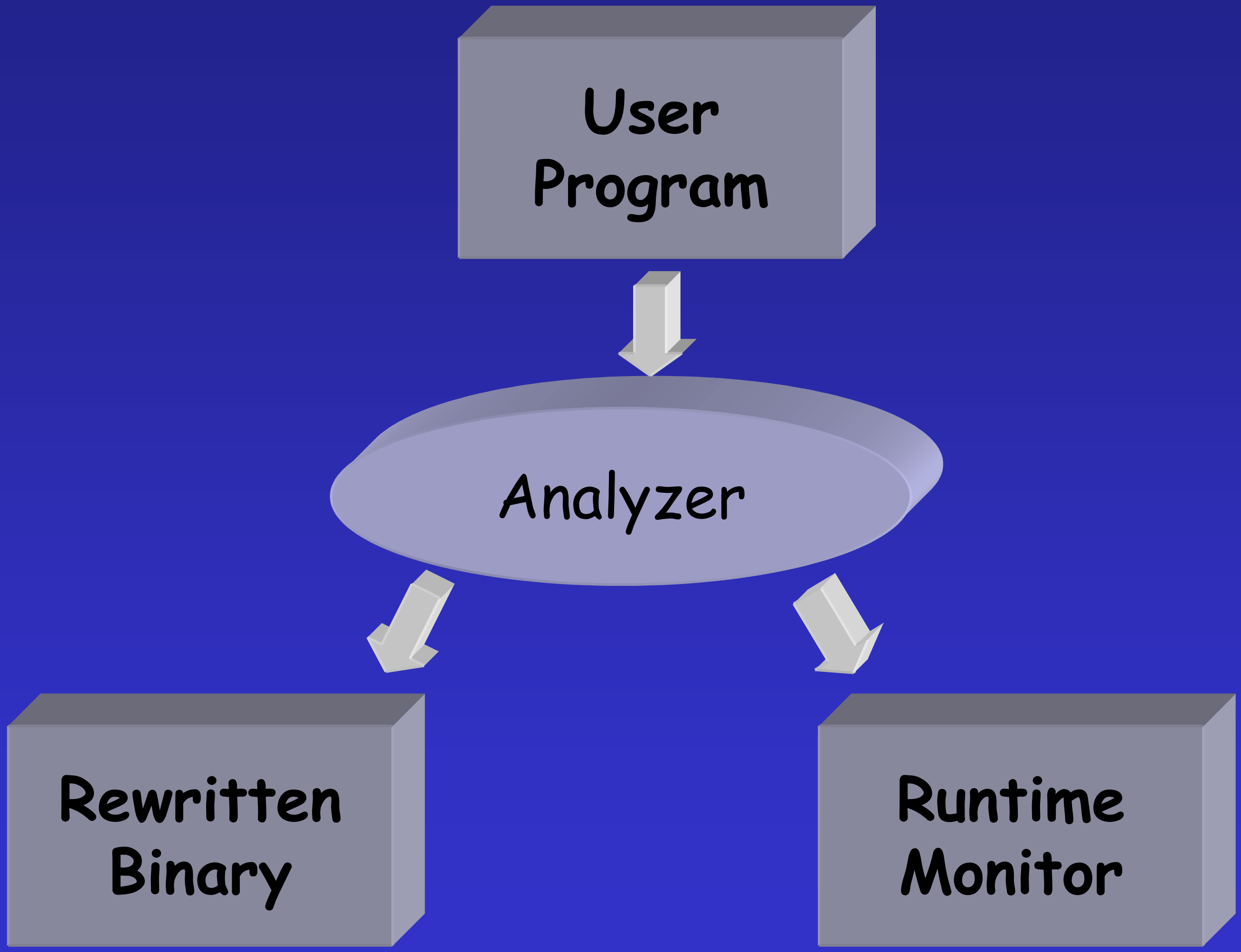


- Snort
  - Detection at service interface: limited to network-based attacks
- Our work
  - Detection at system call interface makes our work independent of intrusion technique

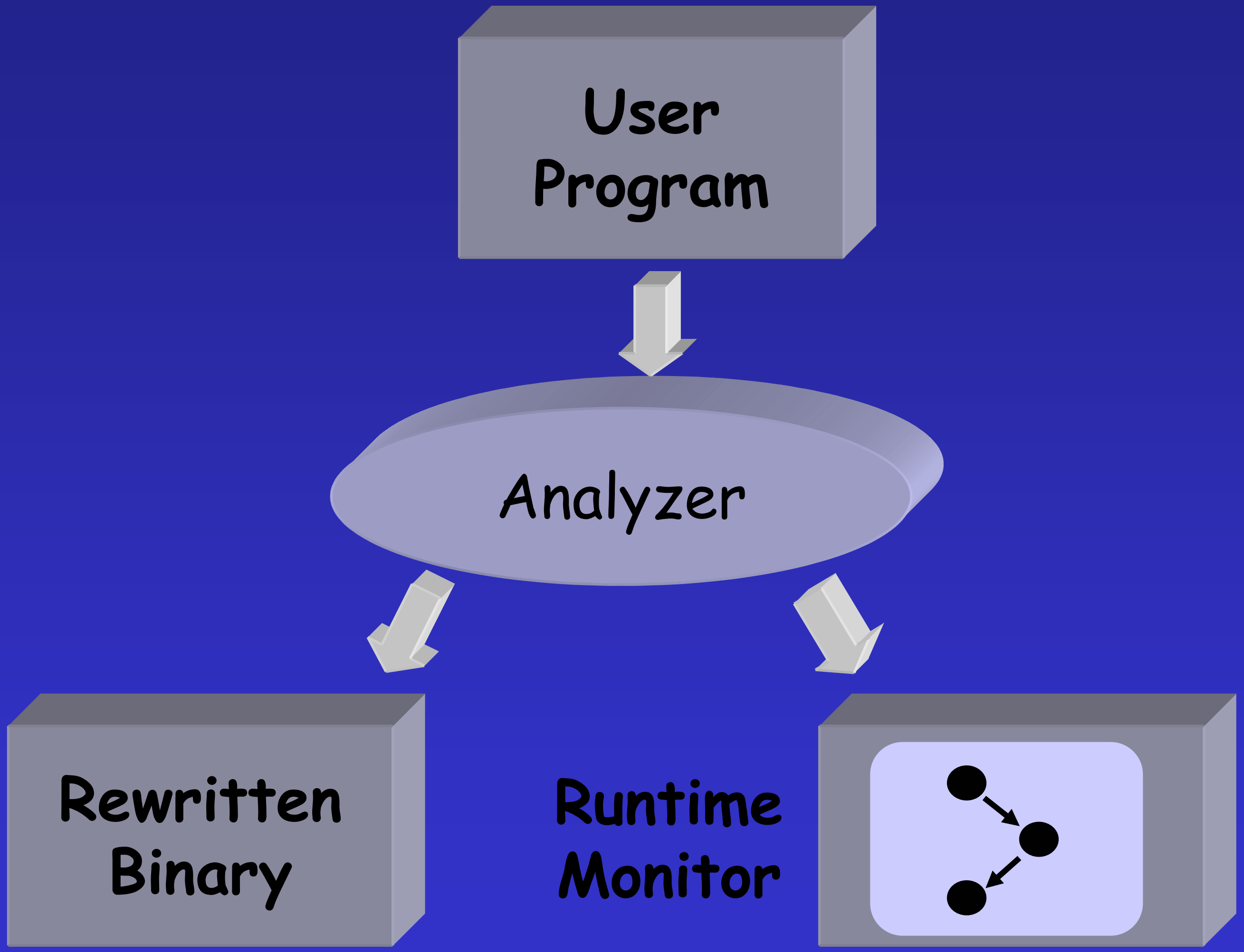
Specification-Based Monitoring

- Specify constraints upon program behavior
  - Static analysis of binary code
  - Construct automaton modeling all system call sequences the program can generate
- Ensure execution does not violate specification
  - Operate the automaton
  - If no valid states, then intrusion attempt occurred

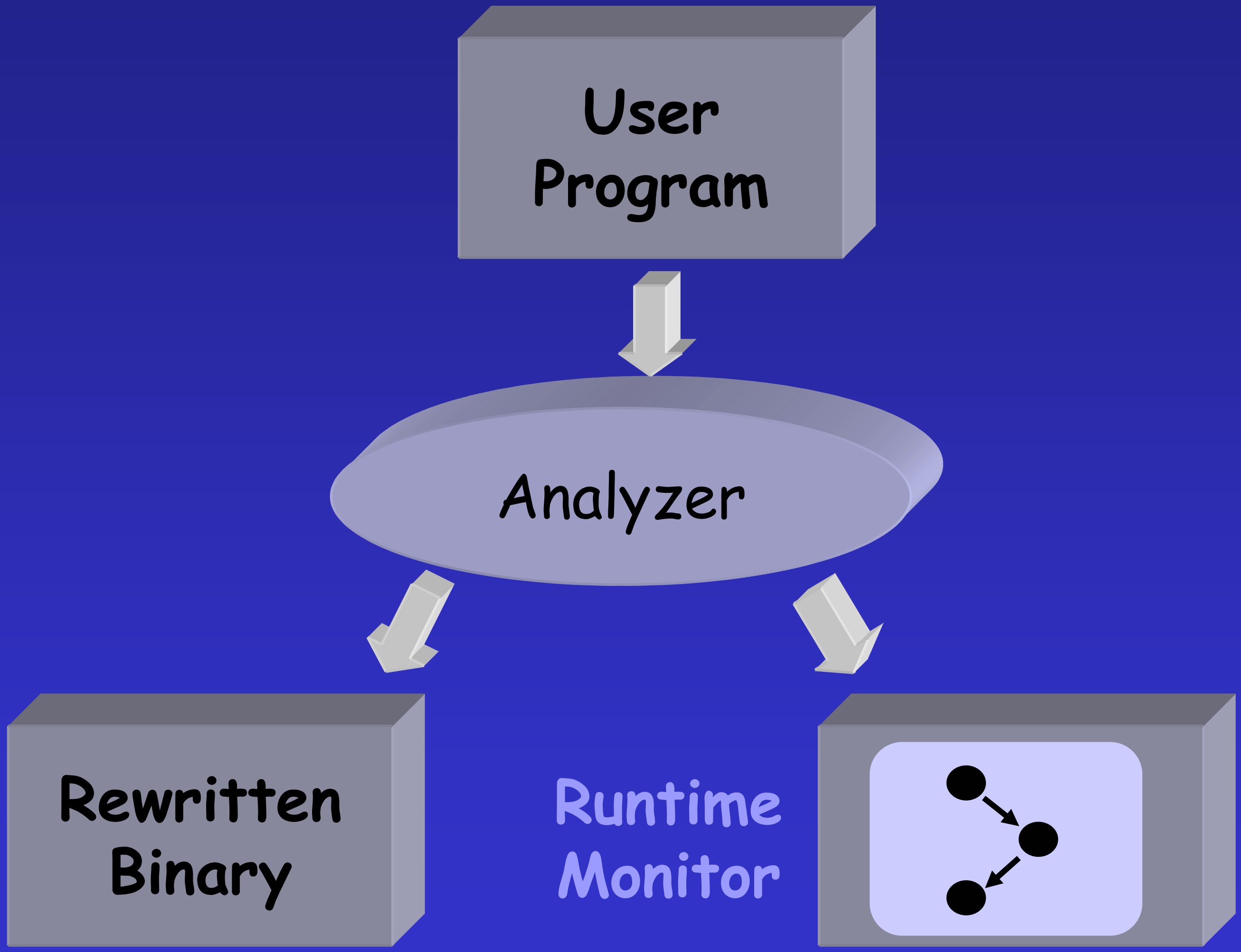
Specification-Based Monitoring



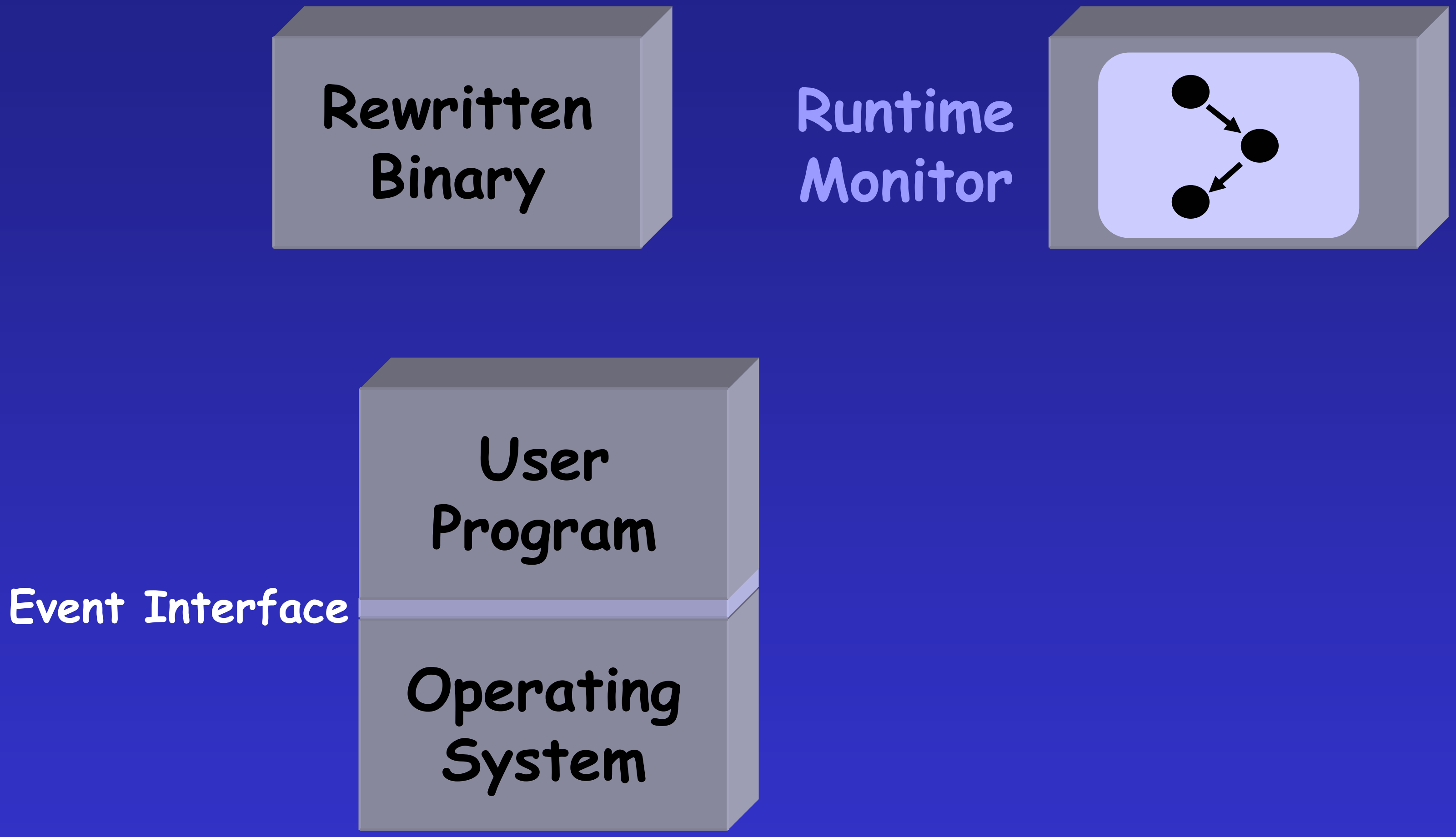
Specification-Based Monitoring



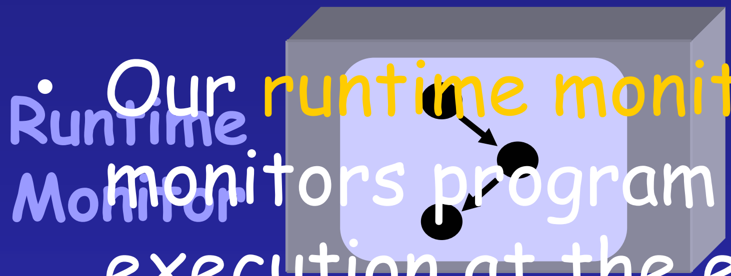
Specification-Based Monitoring



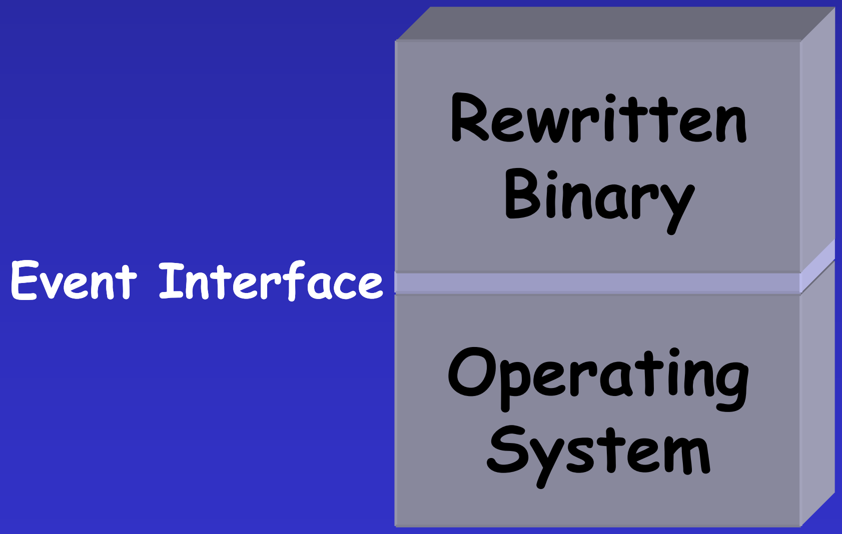
Specification-Based Monitoring



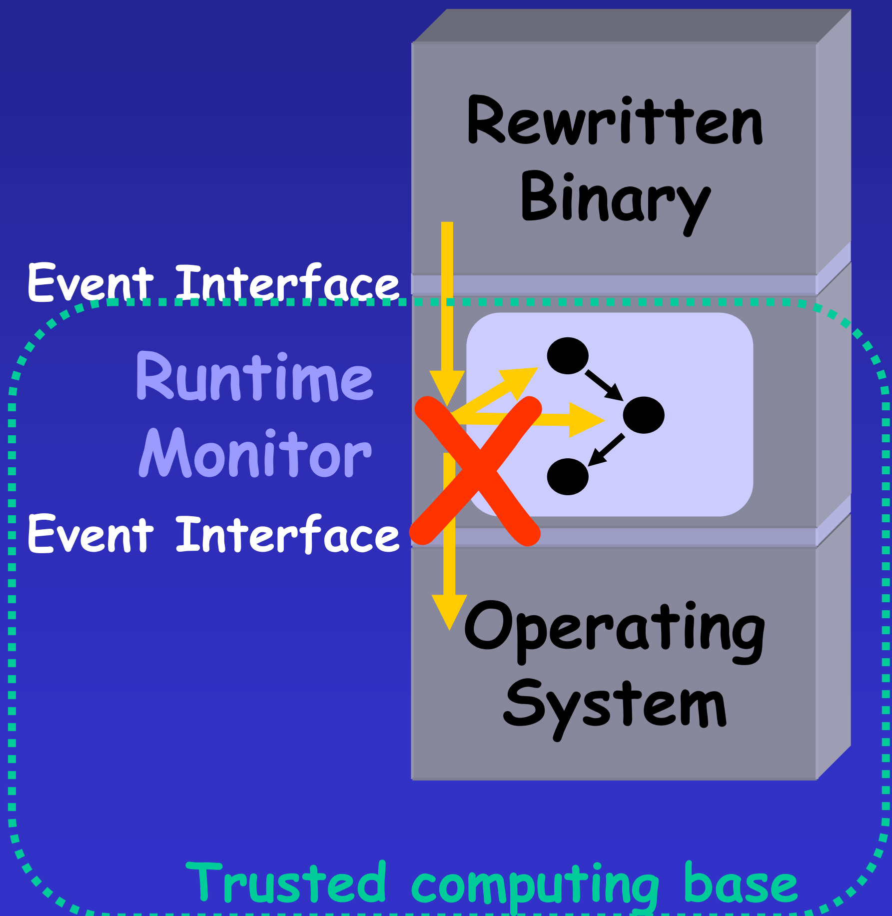
Specification-Based Monitoring



- Our runtime monitor monitors program execution at the event interface layer
- Ensures program events match specification

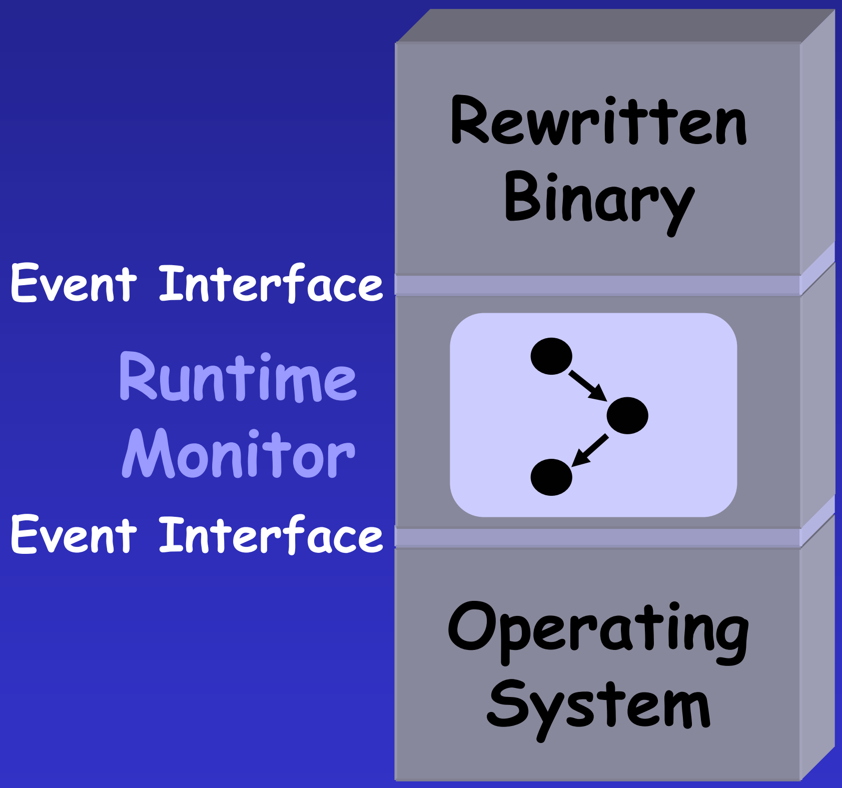


Specification-Based Monitoring



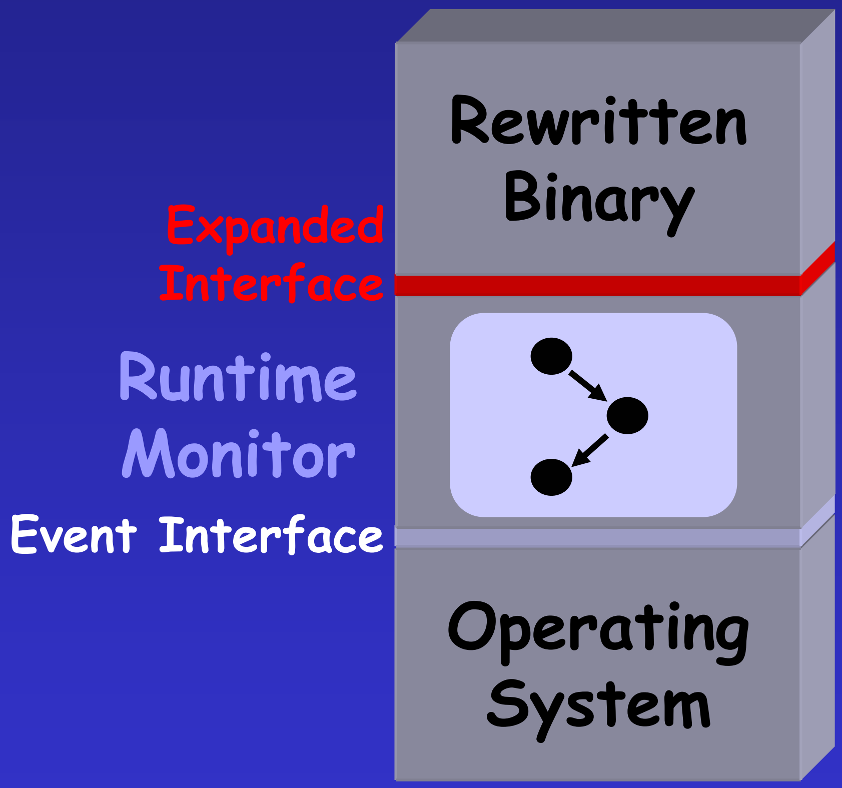
- Our runtime monitor monitors program execution at the event interface layer
- Ensures program events match specification
- Runtime monitor must be part of trusted computing base

Specification-Based Monitoring



- Event interface defines observable events
- Observed events may be superset of system calls
- Expand interface between program and monitor
  - Call-site renaming
  - Null calls

Specification-Based Monitoring

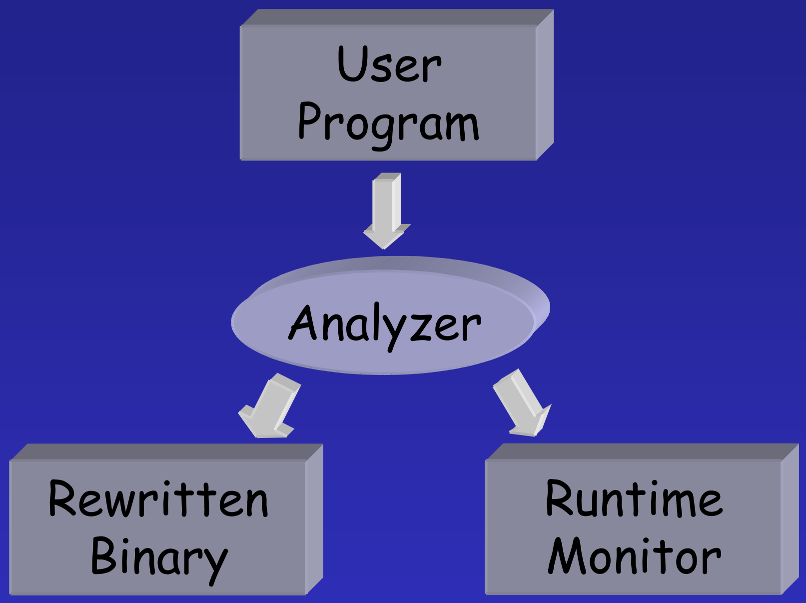


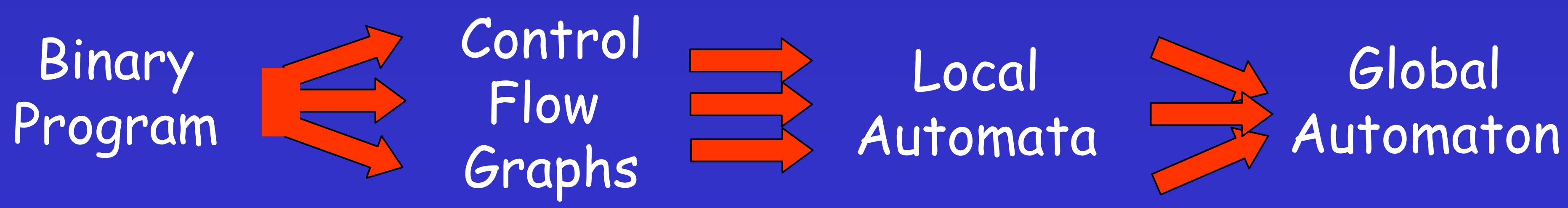
- Expanded set of observable events
  - More precise program modeling
  - More efficient model operation
- User program rewritten to use expanded interface

Observable Events

- Initial interface
  - System call names
  - Add system call arguments
  - Add system call return values
- Expanded interface
  - Add null call names
  - Add null call arguments

Model Construction

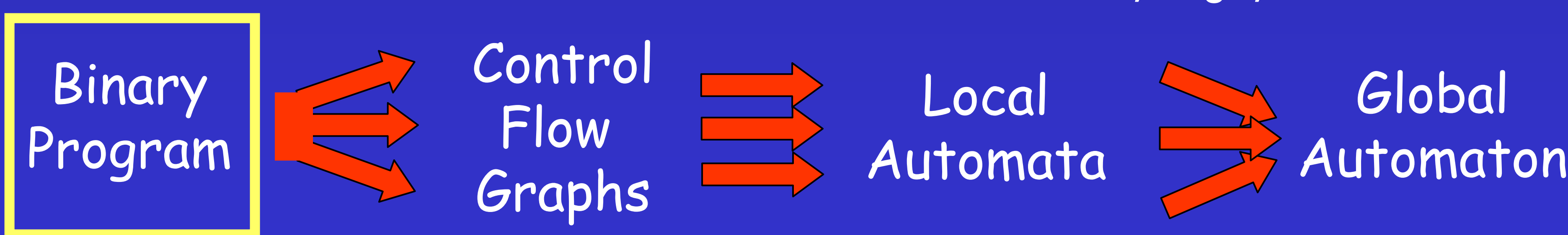




Code Example

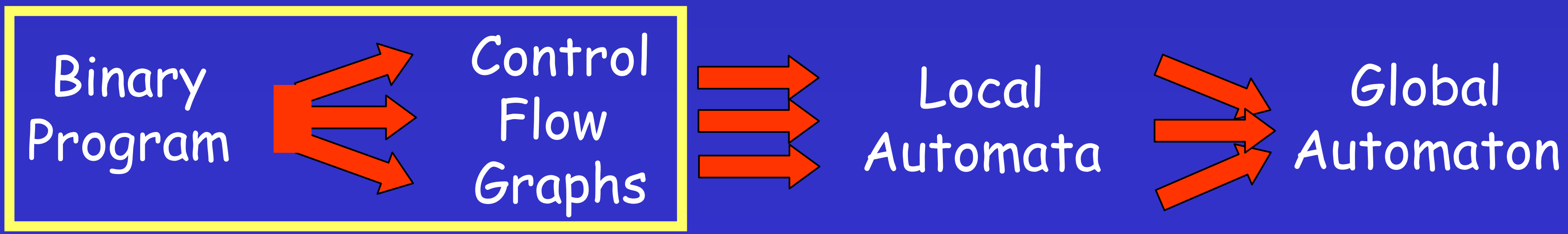
```
int
wrapper (char *file, bool wr)
{
  mode_t mode;
  if (wr)
    mode = O_RDWR;
  else
    mode = O_RDONLY;
  return open(file, mode);
}
```

```
wrapper:
  save %sp, -0x96, %sp
  cmp %i1, 0
  beq,a L1
  mov 1, %o1
  mov 3, %o1
L1:
  call open
  mov %i0, %o0
  ret
  restore %o0, %g0, %o0
```



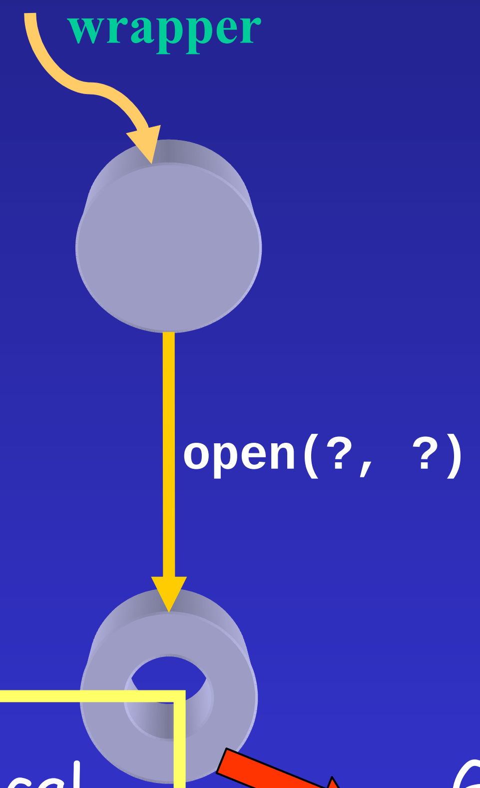
CFG Construction

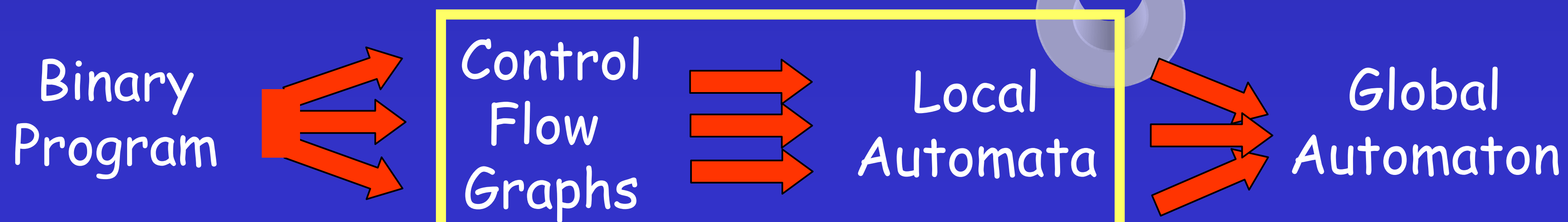
- Executable Editing Library (EEL)
  - Parses SPARC binary programs
  - Identifies functions
  - Constructs control flow graphs (CFG)
  - Rewrites binary program



Local Automaton Construction

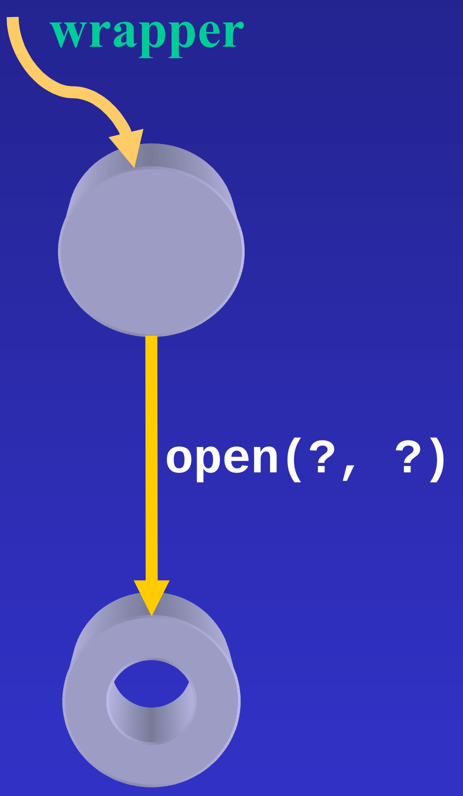
```
int
wrapper (char *file, bool wr)
{
  mode_t mode;
  if (wr)
    mode = O_RDWR;
  else
    mode = O_RDONLY;
  return open(file, mode);
}
```





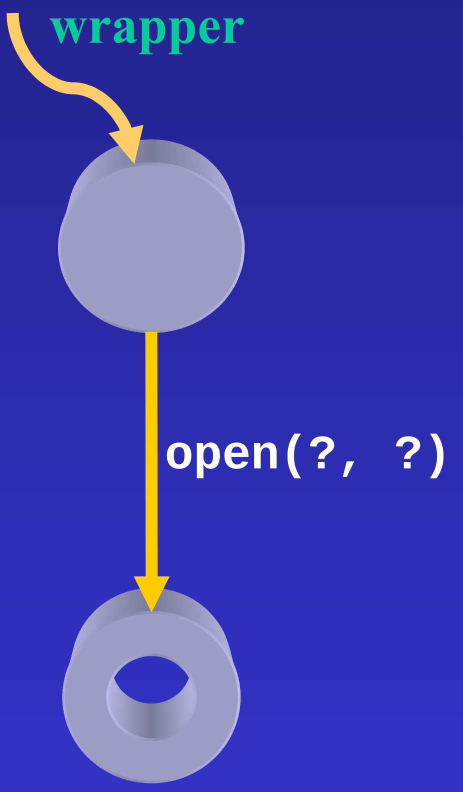
Attack Restriction

- Only sequences of system calls in automata are accepted
  - Attacks that do not match these sequences will fail



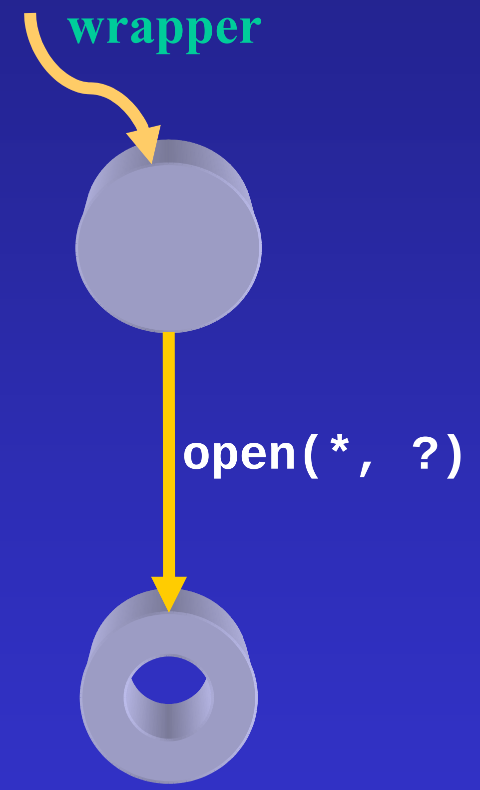
Argument Manipulation

- Shortcoming: attacker can specify any arguments
- Goal: include system call arguments
  - Restricts opportunities for attacker to cause harm
- Argument capture is the analysis technique that statically identifies arguments



Argument Capture

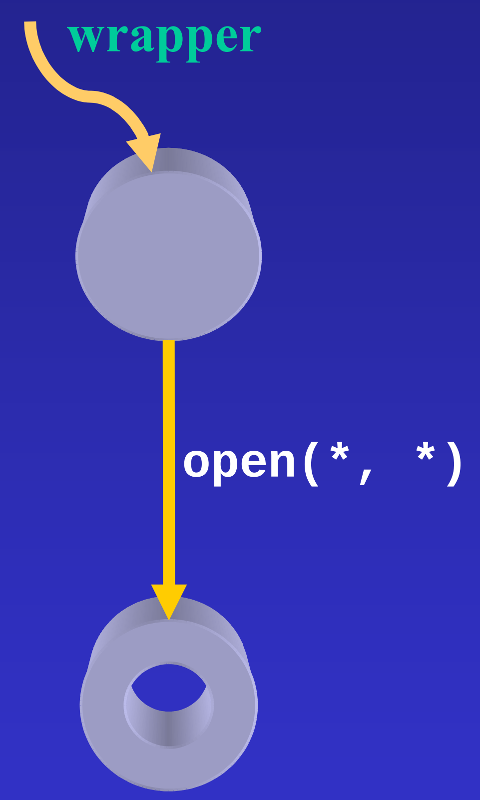
```
int
wrapper (char *file, bool wr)
{
  mode_t mode;
  if (wr)
    mode = O_RDWR
  else
    mode = O_RDONLY;
  return open(file, mode);
}
```



Problem: Interprocedural data flow

Argument Capture

```
int
wrapper (char *file, bool wr)
{
  mode_t mode;
  if (wr)
    mode = O_RDWR;
  else
    mode = O_RDONLY;
  return open(file, mode);
}
```



Problem: Multiple possible values

Improving Data Flow Analysis

- Need for data flow analysis
  - Argument capture
  - Identifying targets of indirect transfers
- EEL provides backward register slicing

Improving Data Flow Analysis

EEL

- Intraprocedural
- Tree based
- Recovers constants

Improving Data Flow Analysis

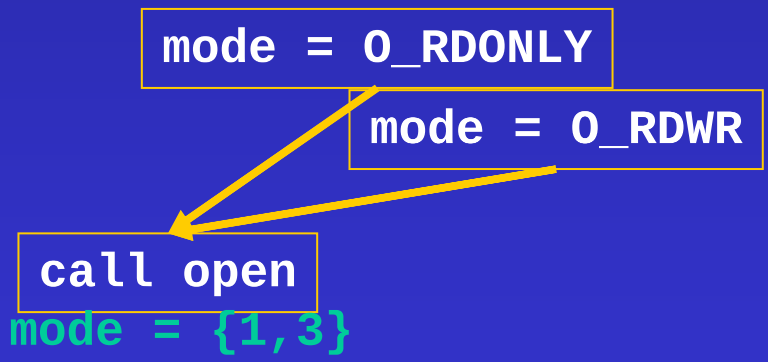
- New infrastructure component: data dependence graph
  - Subgraph of program dependence graph
  - Collection of expression graphs that set register values
  - Includes interprocedural data flows

Data Dependence Graph

- Nodes are individual instructions
- 3 edge types connect instructions I→J:
  - Intraprocedural: I writes a data value that J reads
  - Call arguments: I sets an actual argument to a call, J reads the formal argument
  - Call return: I sets a call return value, J reads the return value

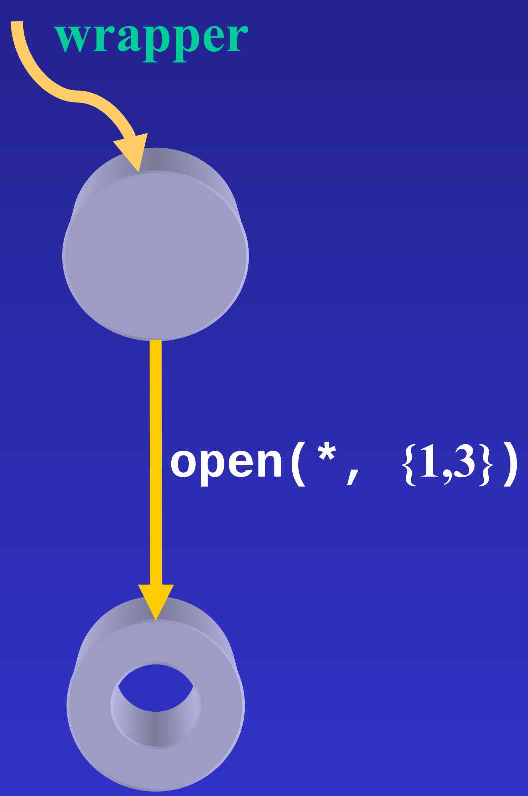
DDG Construction

```
int
wrapper (char *file, bool wr)
{
  mode_t mode;
  if (wr)
    mode = O_RDWR;
  else
    mode = O_RDONLY;
  return open(file, mode);
}
```



Argument Capture

```
int
wrapper (char *file, bool wr)
{
  mode_t mode;
  if (wr)
    mode = O_RDWR;
  else
    mode = O_RDONLY;
  return open(file, mode);
}
```



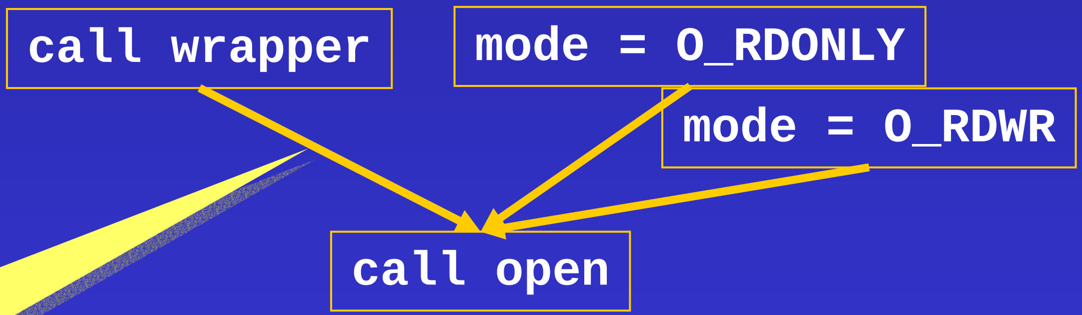
Code Example

```
int
wrapper (char *file, bool wr)
{
  mode_t mode;
  if (wr)
    mode = O_RDWR;
  else
    mode = O_RDONLY;
  return open(file, mode);
}
```

```
void
program (char *file)
{
  int fd1, fd2;
  char *name = “myfile”;

  fd1 = wrapper(name, true);
  fd2 = wrapper(“/etc/passwd”,
                false);
  close(fd1);
  close(fd2);
}
```

DDG Construction

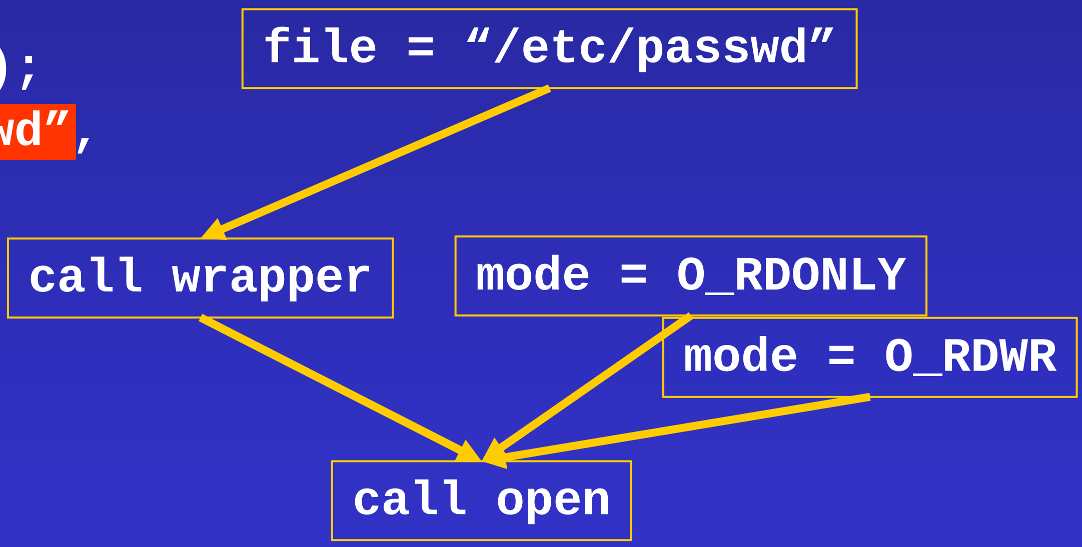
```
int
wrapper (char *file, bool wr)
{
  mode_t mode;
  if (wr)
    mode = O_RDWR;
  else
    mode = O_RDONLY;
  return open(file, mode);
}
```



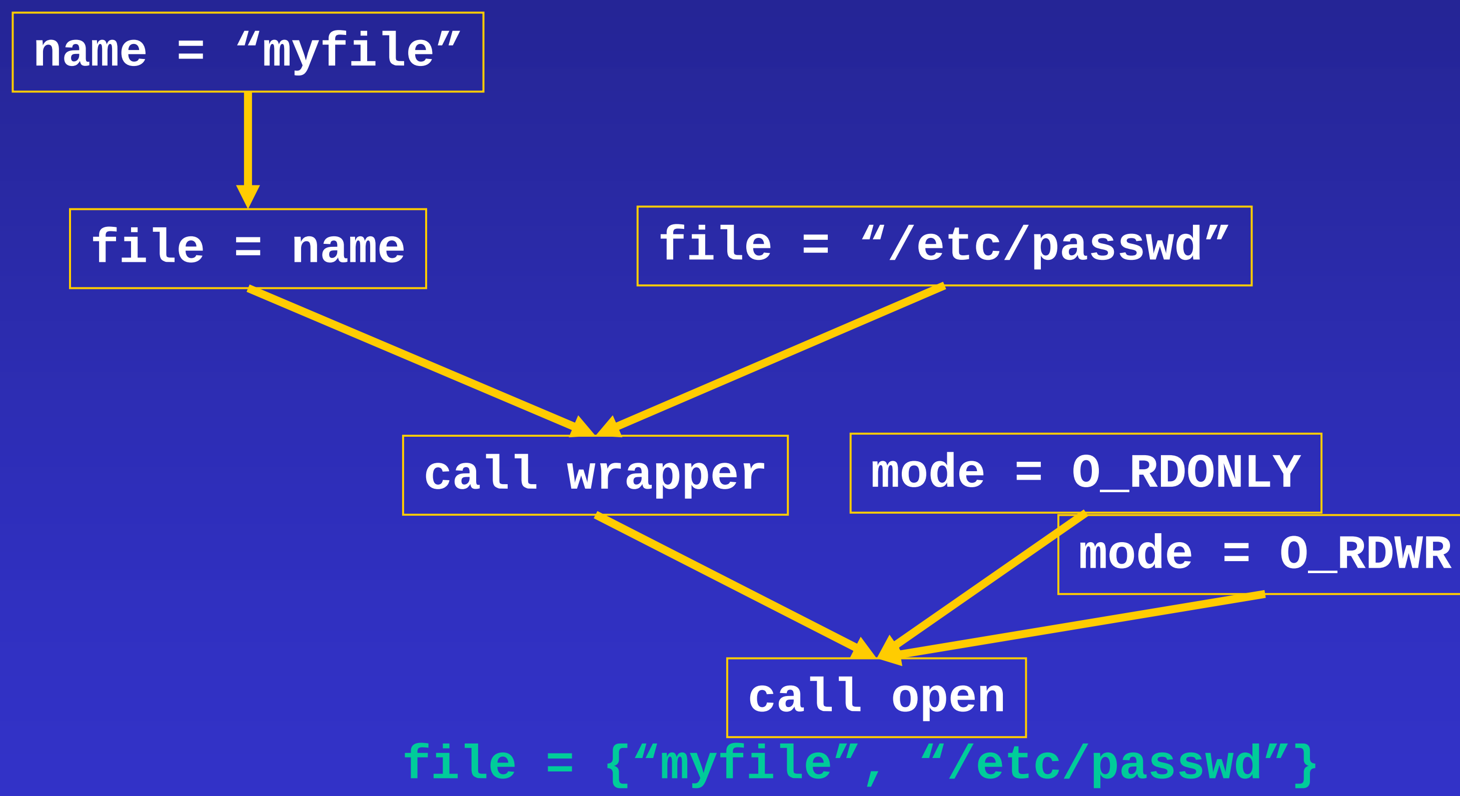
Interprocedural edge

DDG Construction

```
void
program (char *file)
{
   int fd1, fd2;
   char *name = "myfile";

   fd1 = wrapper(name, true);
   fd2 = wrapper("/etc/passwd",
                 false);
   close(fd1);
   close(fd2);
}
```

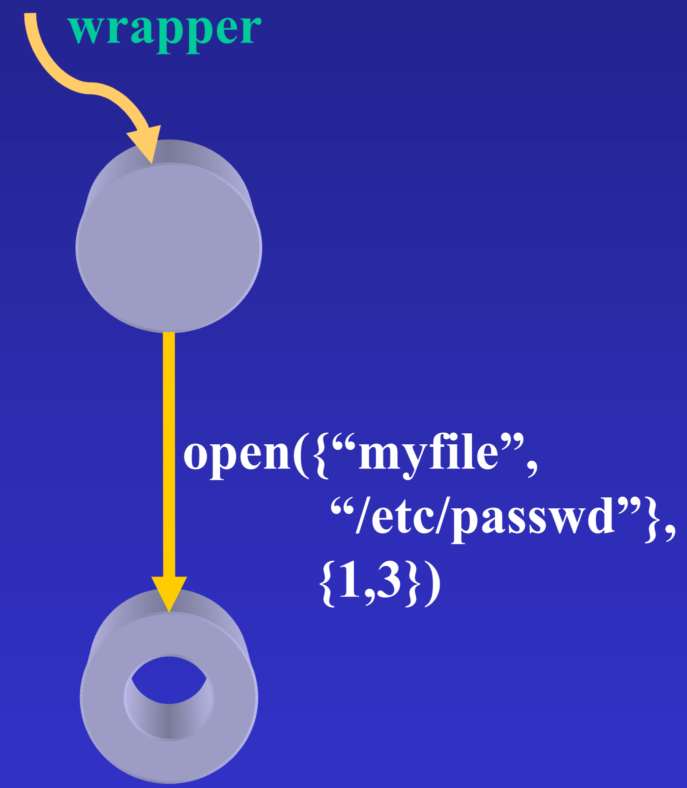


DDG Construction



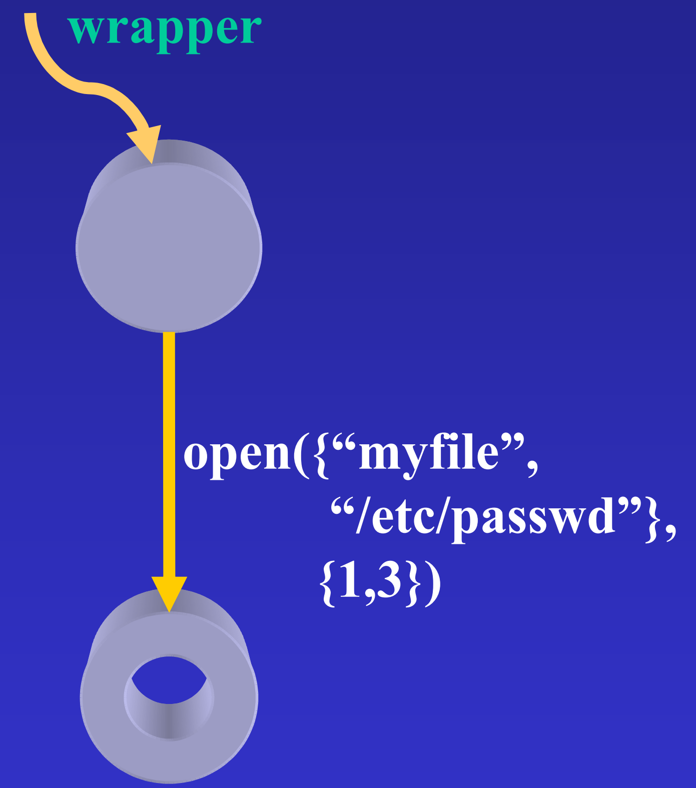
Argument Capture

```
int
wrapper (char *file, bool wr)
{
  mode_t mode;
  if (wr)
    mode = O_RDWR;
  else
    mode = O_RDONLY;
  return open(file, mode);
}
```



Argument Capture

- Goal reached
  - All arguments recovered



Data Dependence Graph

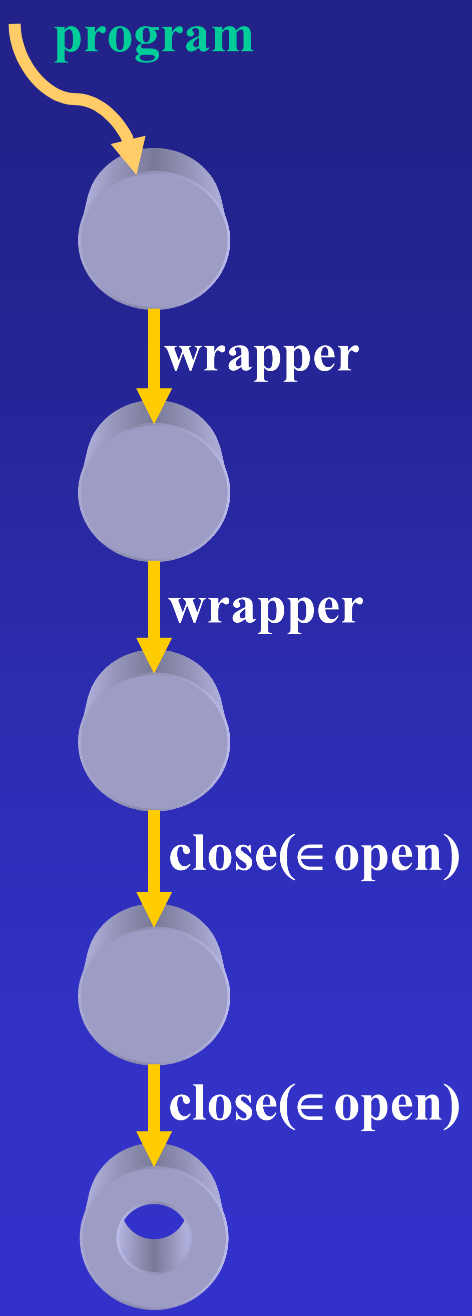
- Foundation of all data flow analyses used by the binary analyzer
  - Identification of indirect transfer targets
    - Call graph construction
  - Argument capture

Improving Argument Capture

- Using EEL's analysis, argument capture treated each value as an integer
- We can do more: general notion of argument types
  - Integer
  - Set of integers
  - Return values of previous system calls
  - Regular expressions for string arguments
  - Arguments passed to library functions

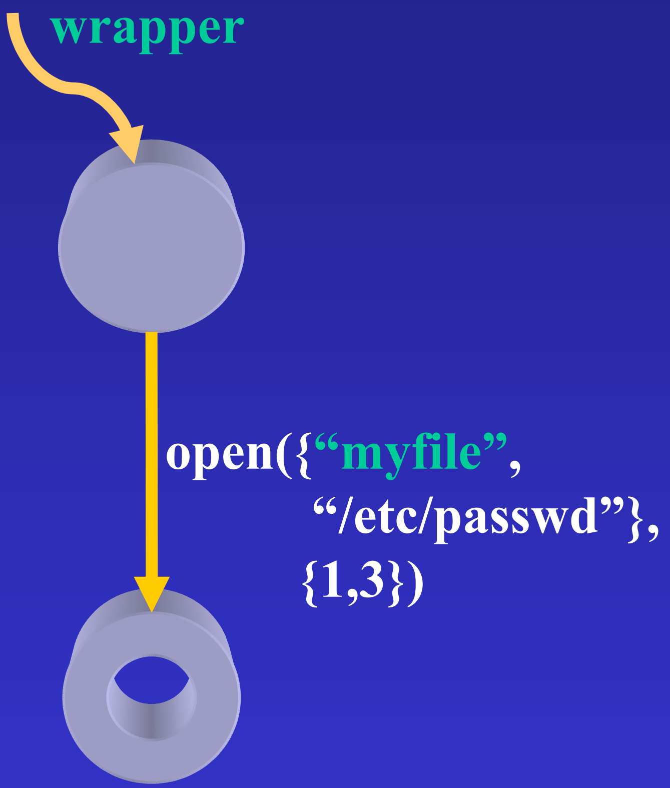
Syscall Return Values

```
void
program (char *file)
{
  int fd1, fd2;
  char *name = “myfile”;

  fd1 = wrapper(name, true);
  fd2 = wrapper(“/etc/passwd”,
                  false);
  close(fd1);
  close(fd2);
}
```



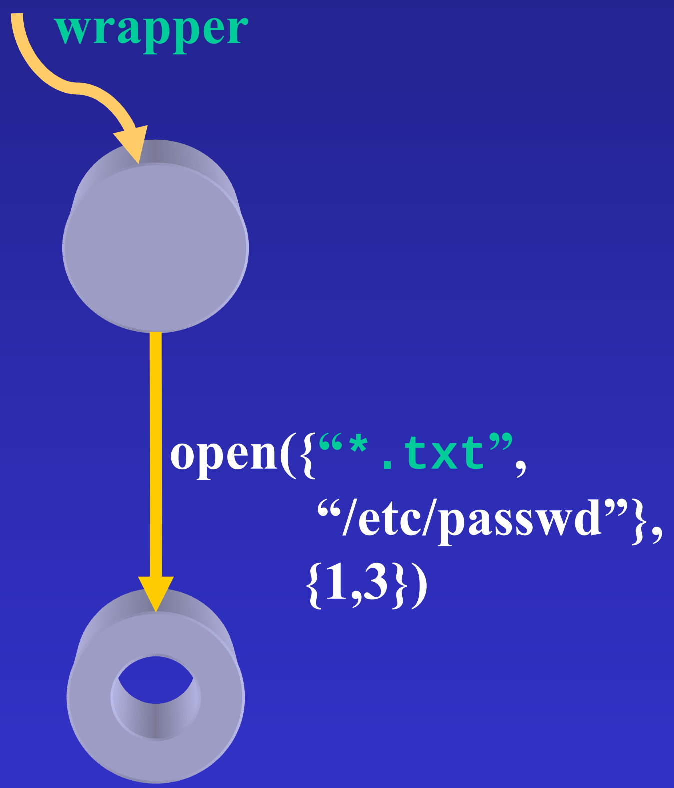
Regular Expressions

```
void
program (char *file)
{
  int fd1, fd2;
  char *name = "myfile";

  fd1 = wrapper(name, true);
  fd2 = wrapper("/etc/passwd",
                false);
  close(fd1);
  close(fd2);
}
```



Regular Expressions

```
void
program (char *file)
{
  int fd1, fd2;
  char *name =
       strcat(file, ".txt");
  fd1 = wrapper(name, true);
  fd2 = wrapper("/etc/passwd",
                  false);
  close(fd1);
  close(fd2);
}
```



Improving Argument Capture

EEL

- Intraprocedural
- Tree based
- Recovers constants

Data Dependence Graph

- Interprocedural
- Graph based
- Recovers constants, sets, return values, regular expressions, and arguments

Measurements

- Number of arguments recovered
  - Intraprocedural, constants
  - Interprocedural, constants
  - Interprocedural, general representation

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

Results

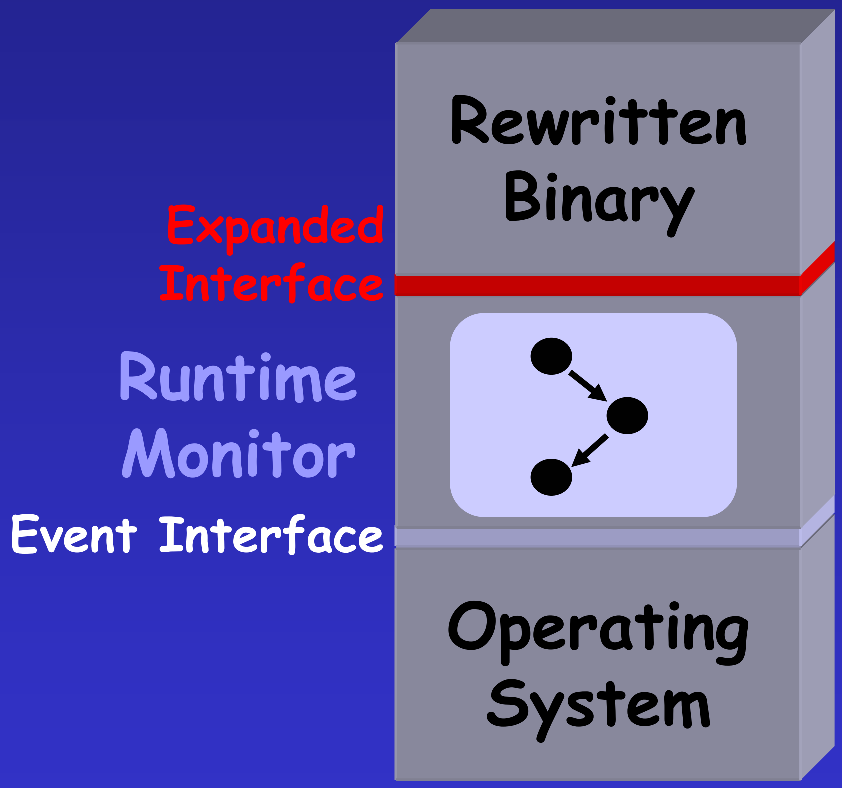
- Number of recovered arguments:

Technique	gzip	finger	procmail
Intraprocedural, constant	30	149	206
Interprocedural, constant	50	157	212
Interprocedural, general	81	227	271

Improving Argument Capture

- Restricts the attacker because the specification limits acceptable arguments

Smart Null Call Insertion



- Null calls are dummy system calls
  - Part of the expanded interface
  - Used by the monitor to update the model
  - Do not cross the interface to the operating system

Smart Null Call Insertion

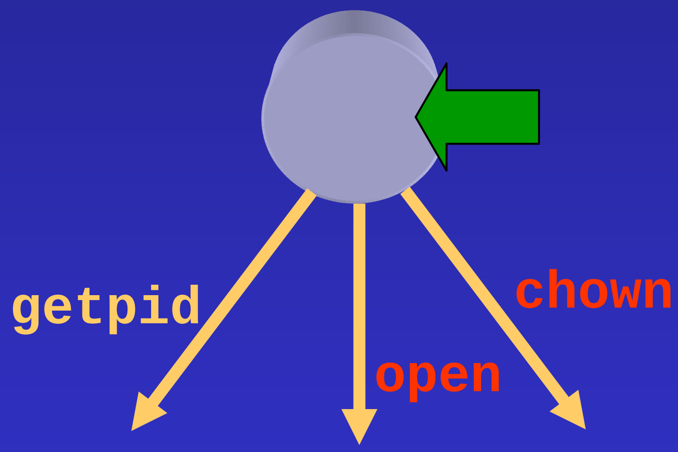
- Previous experiments have shown:
  - Null calls improve precision
    - Restrict paths followed in automaton
  - Null calls improve performance of push-down automaton models
- Prior work used naïve null call placement
  - Based solely upon the fan-in of the function
  - Fan-in is a crude approximation of precision gain

Smart Null Call Insertion

- New algorithm places null calls based upon expected precision gain
- Expected effect:
  - Greater precision gains at less cost

Smart Null Call Insertion

- Precision metric: average branching factor



- Lower values indicate greater precision

Smart Null Call Insertion

- Statically compute branching factor at function entry points
- Instrument functions that contribute most to average branching factor

- Effect: See best precision improvement with fewest null call insertions

Smart Null Call Insertion

- Measurements: Comparison to naïve insertion
  - Monitoring overhead
  - Precision gain versus null call overhead

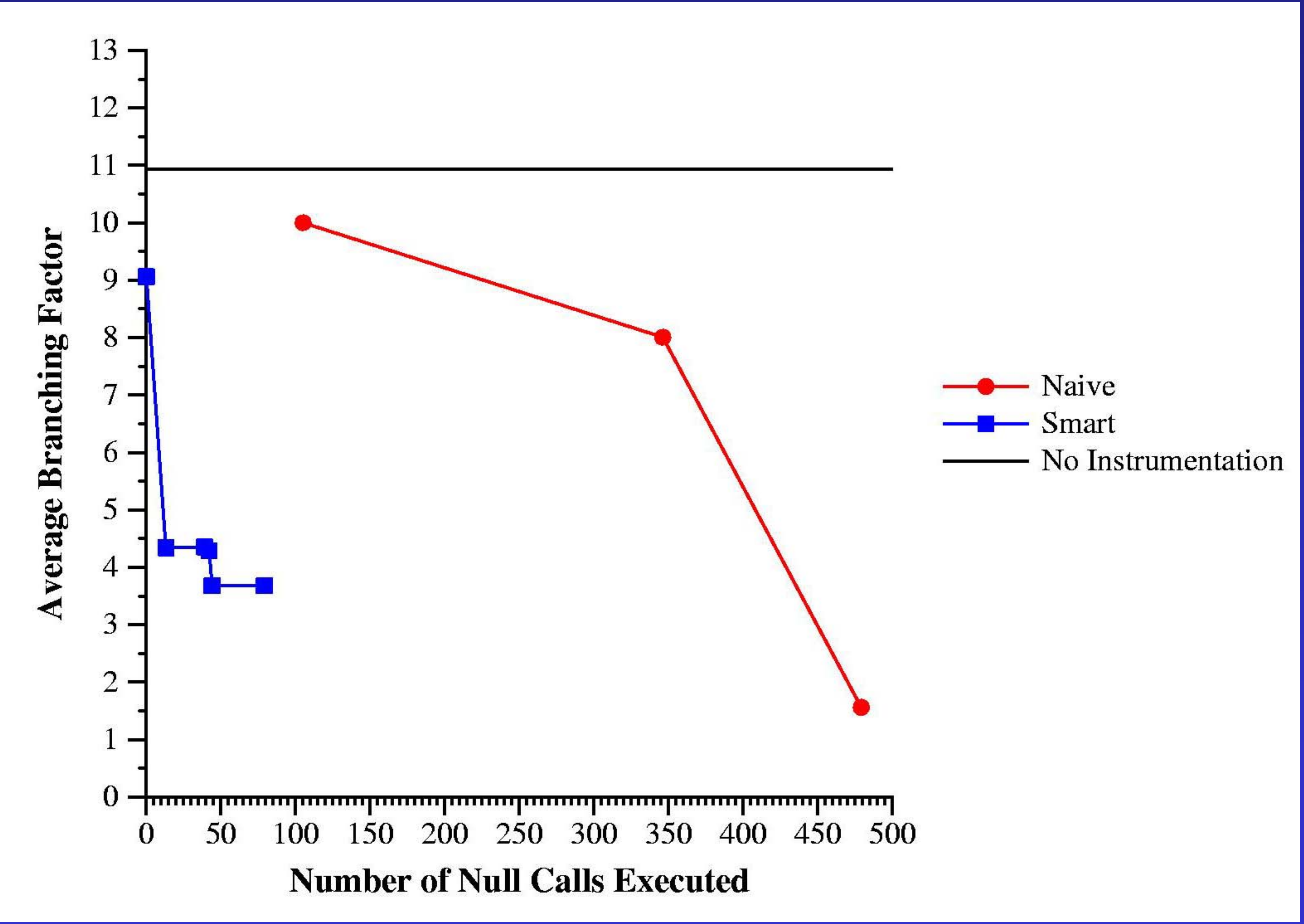
Null Calls Monitoring Overhead

- Naïve insertion:

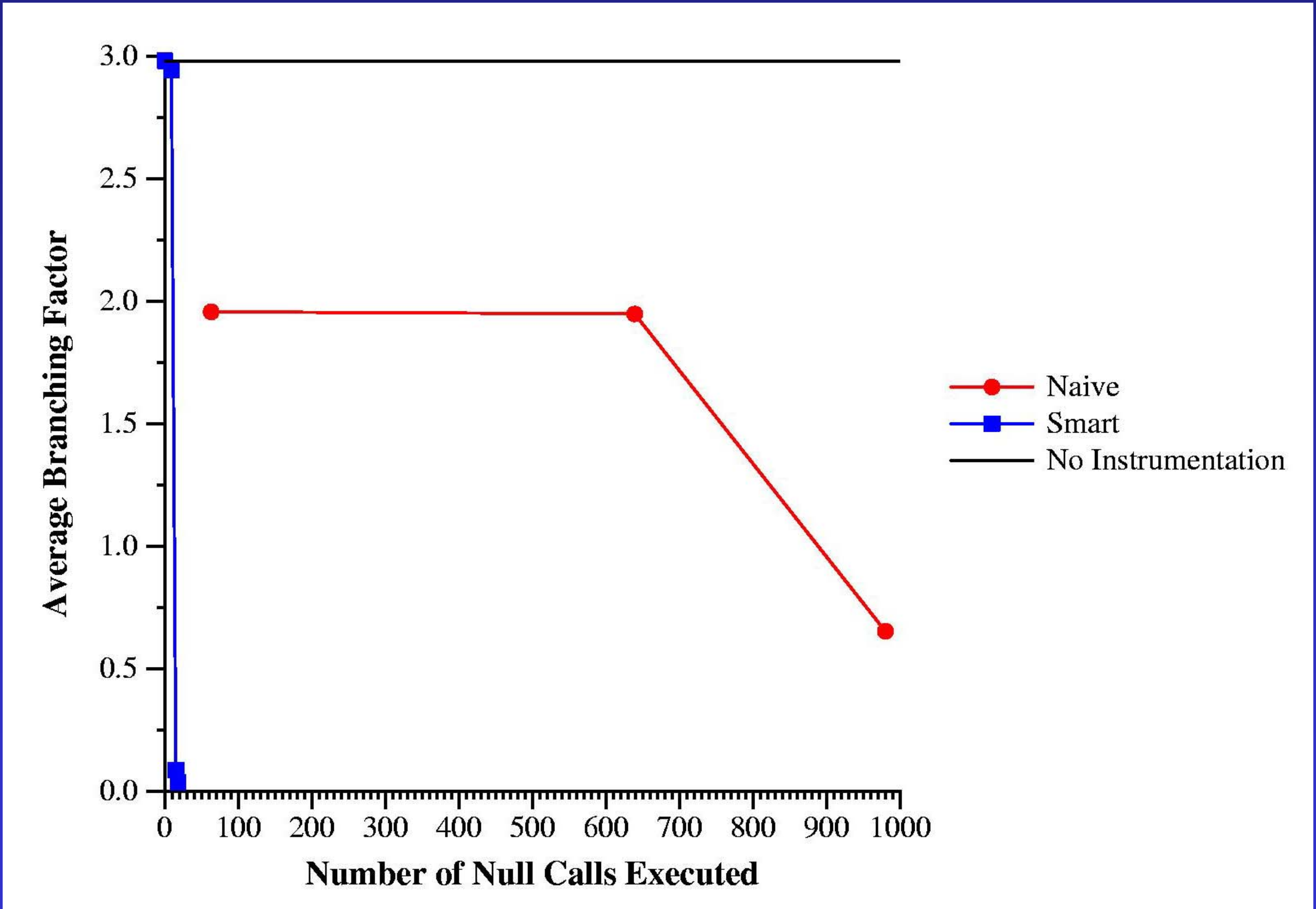
Insertion Rate	High	Medium	Low
gzip	747.0 %	< 0.1 %	< 0.1 %
GNU finger	0.1 %	0.1 %	< 0.1 %
procmail	0.8 %	1.1 %	0.7 %

- Intelligent insertion:
  - Miniscule! Lost in measuring noise.

Null Call Precision vs. Overhead: procmail



Null Call Precision vs. Overhead: finger



Important Ideas

- We develop specification-based monitoring techniques using static binary analysis to detect intrusions.
- Interprocedural slicing improves argument capture by including data flow information.
- Intelligent null call insertion maximizes precision gain with minimal performance impact.

Technical Agenda

- Integrating other specification sources
- C++ vtable analysis
- Intelligent null call insertion
- Interprocedural data flow analysis
- General argument representation

Technical Agenda

- Integrating other specification sources
- C++ vtable analysis
- Construction of accurate models for dynamically linked applications
- Abstract stack representations for PDA models

Specification-Based Monitoring: Improving Model Precision

Jonathon Giffin, Somesh Jha, Barton Miller
University of Wisconsin
Wisconsin Safety Analyzer