Efficient Context-Sensitive Intrusion Detection

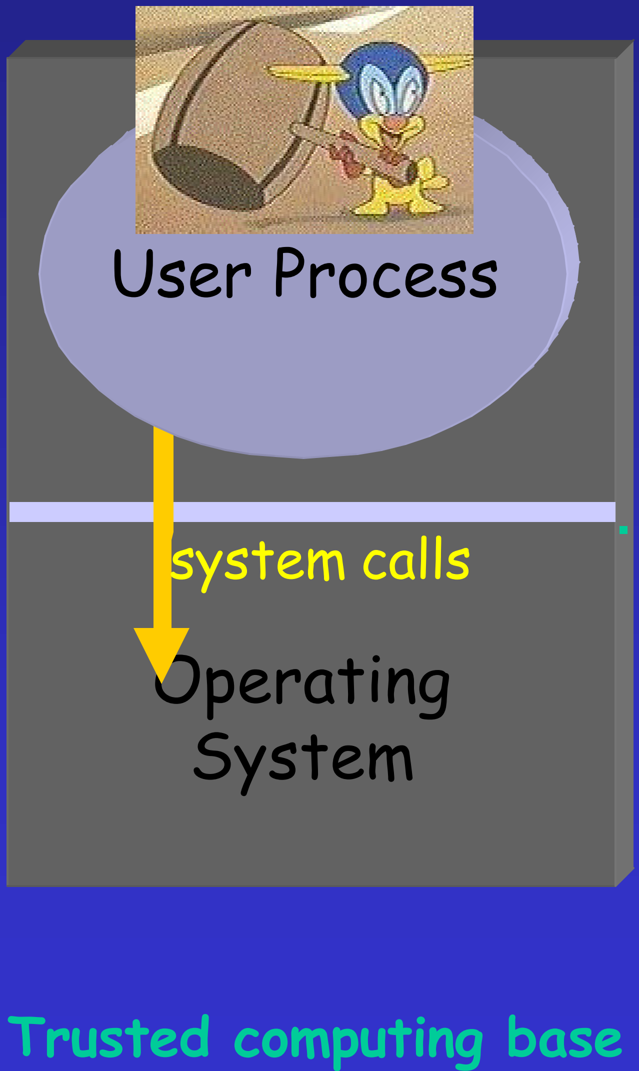
Jonathon T. Giffin
Somesh Jha, Barton P. Miller
University of Wisconsin
{giffin,jha,bart}@cs.wisc.edu

WiSA - Wisconsin Safety Analyzer
http://www.cs.wisc.edu/wisa

Model-Based Intrusion Detection

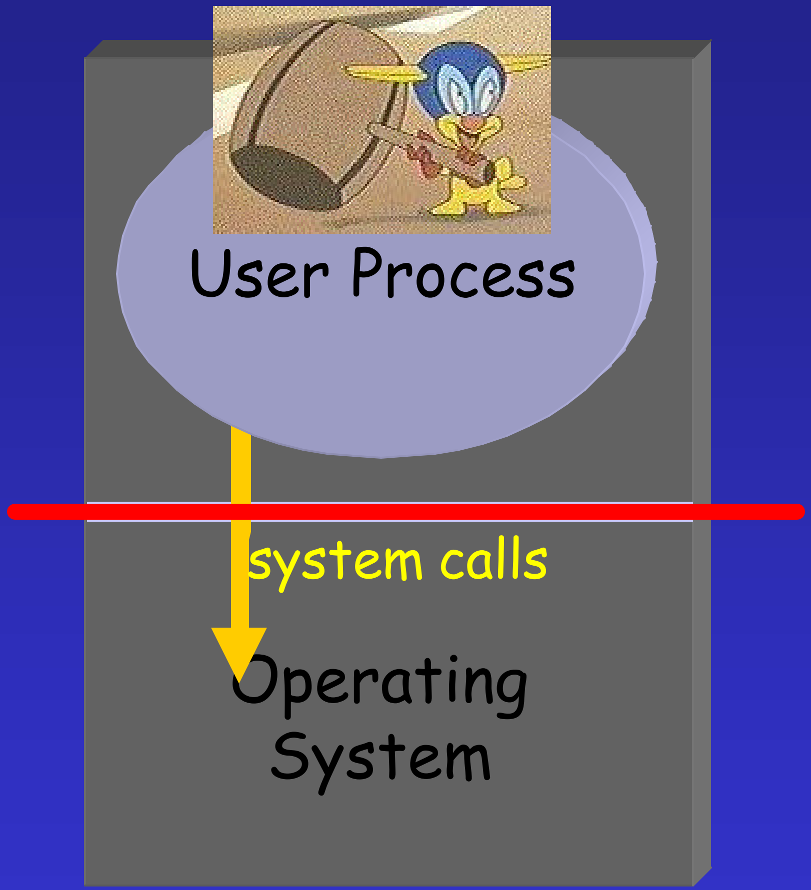
- Constructing program models using static binary analysis
- Accuracy/performance tradeoff in prior models
- Our new Dyck model solves tradeoff
- Data-flow analysis to recover arguments

Worldview



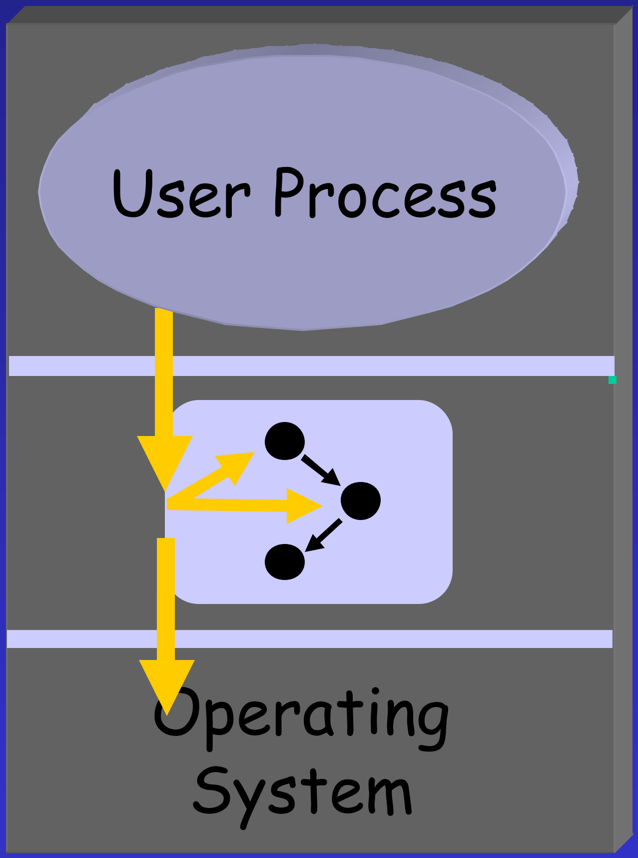
- Running processes make operating system requests
- Changes to trusted computing base done via these requests
- Attacker subverts process to generate malicious requests

Our Objective



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

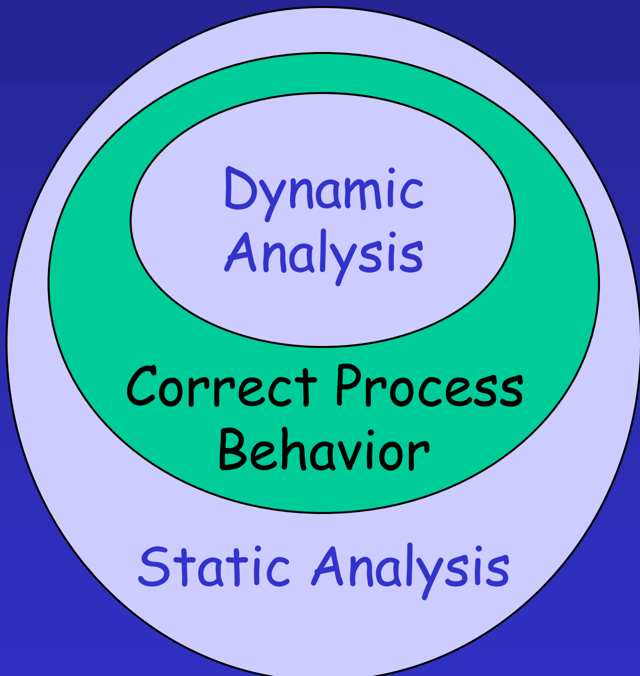
Model-Based Intrusion Detection



Trusted computing base

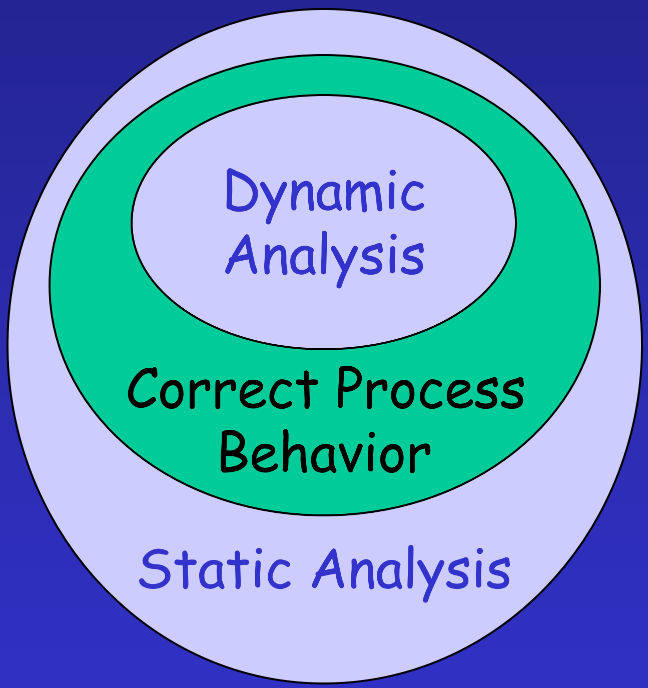
- Build model of correct program behavior
- Runtime monitor ensures execution does not violate model
- Runtime monitor must be part of trusted computing base

Automated Model Construction



- Dynamic analysis
  - Under-approximates correct behavior
  - False alarms
  - Forrest, Sekar, Lee
- Static analysis
  - Over-approximates correct behavior
  - False negatives
  - Wagner&Dean, our work
  - Previous attempts at precise models problematic

Automated Model Construction

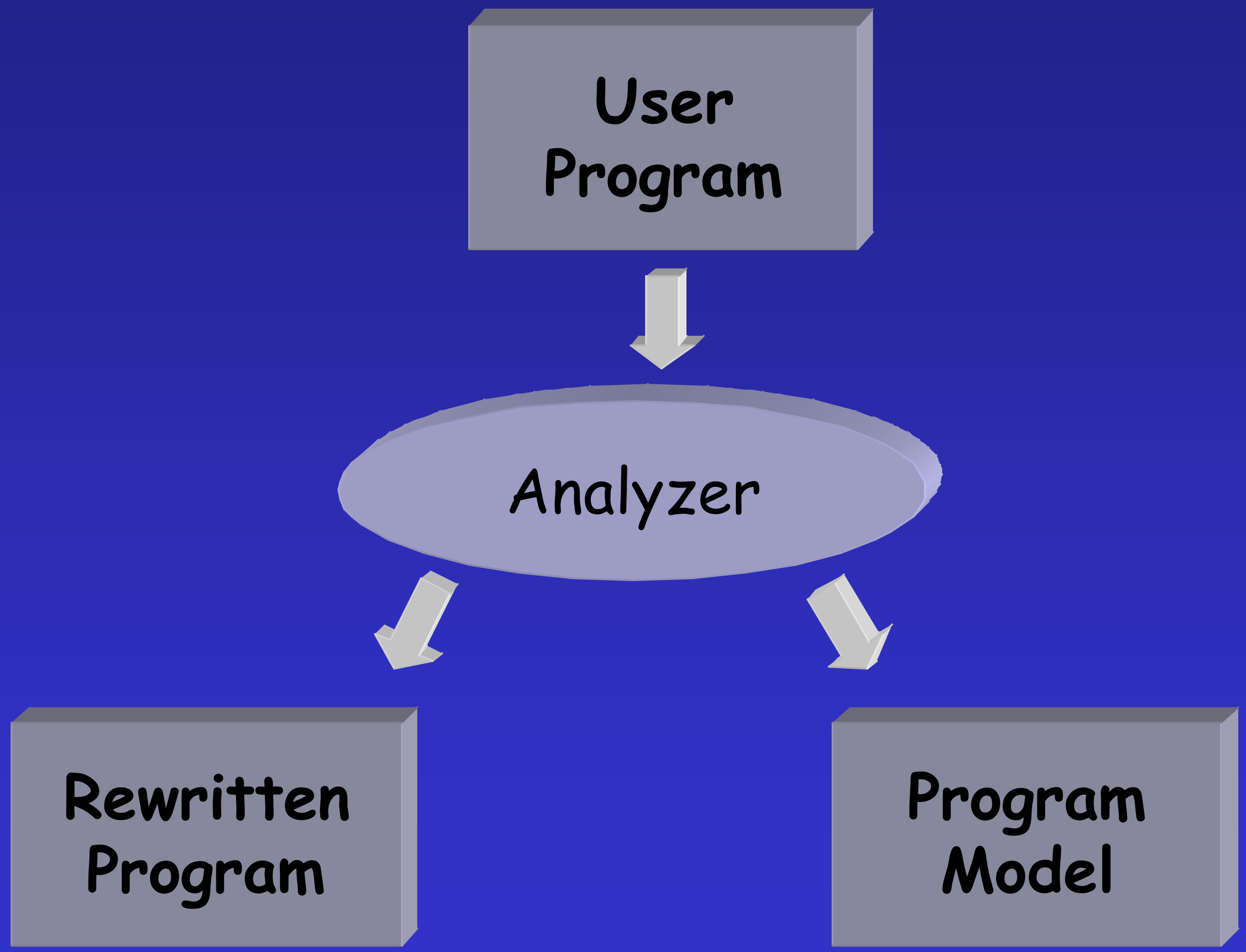


- Static analysis challenge
  - Design an efficient, context-sensitive model
- Techniques
  - Dyck model
  - Argument recovery

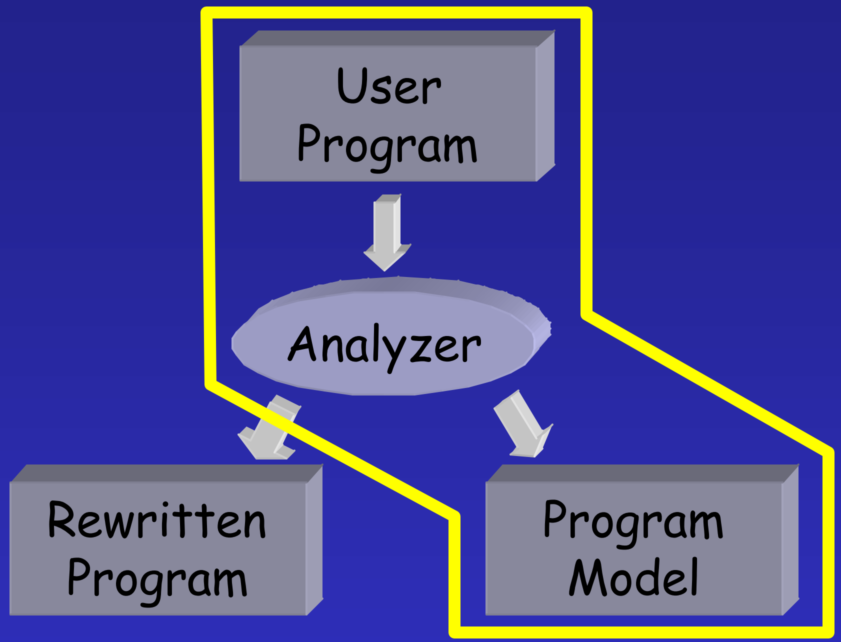
Our Approach

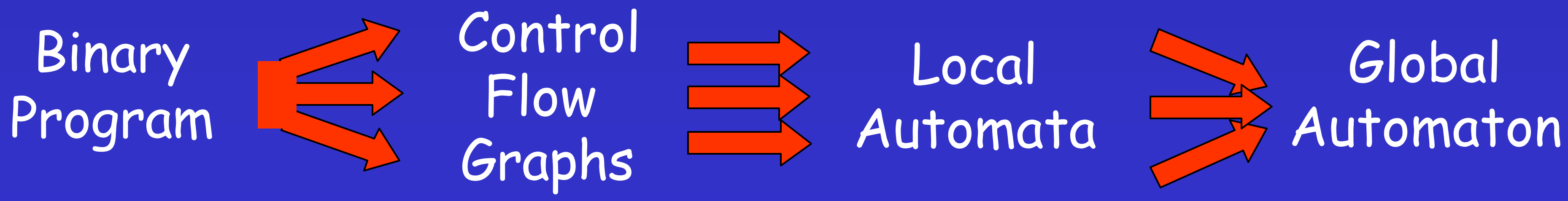
- Build model of correct program behavior
  - Static analysis of binary code
  - Construct an automaton modeling all system call sequences the program can generate
- Ensure execution does not violate model
  - Use automaton to monitor system calls.
  - If automaton reaches an invalid state, then an intrusion attempt occurred.

Program Analysis

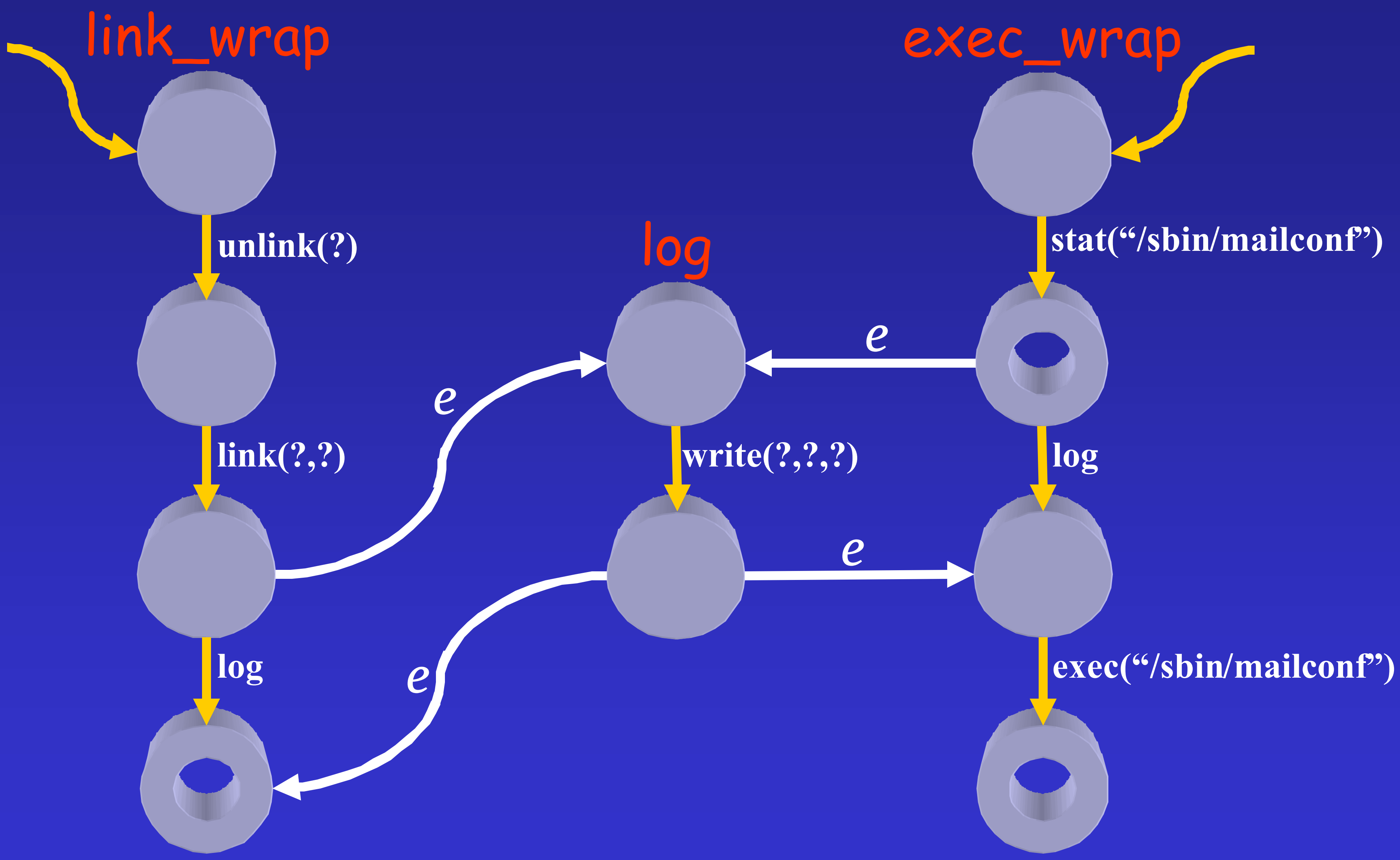


Model Construction





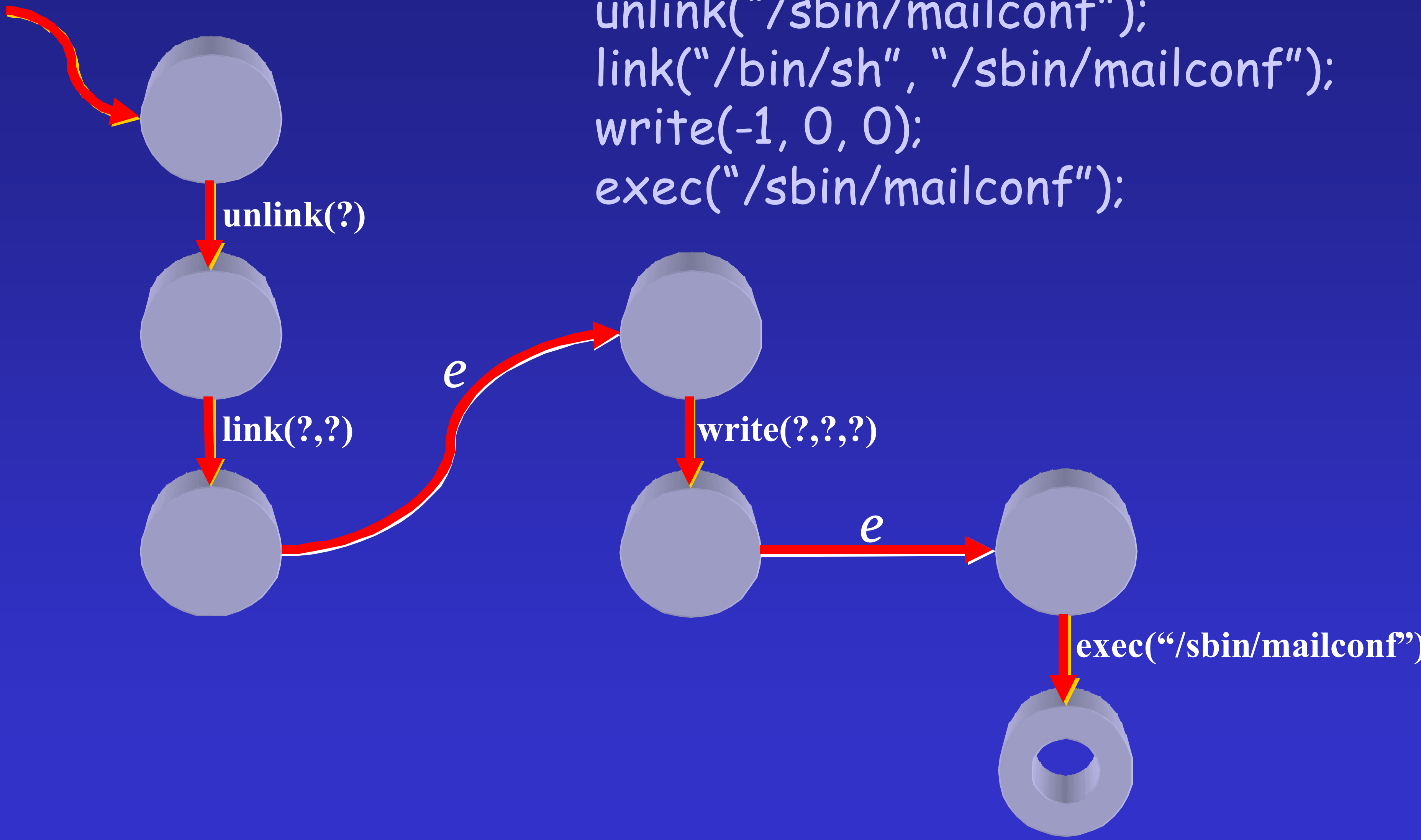
NFA Model



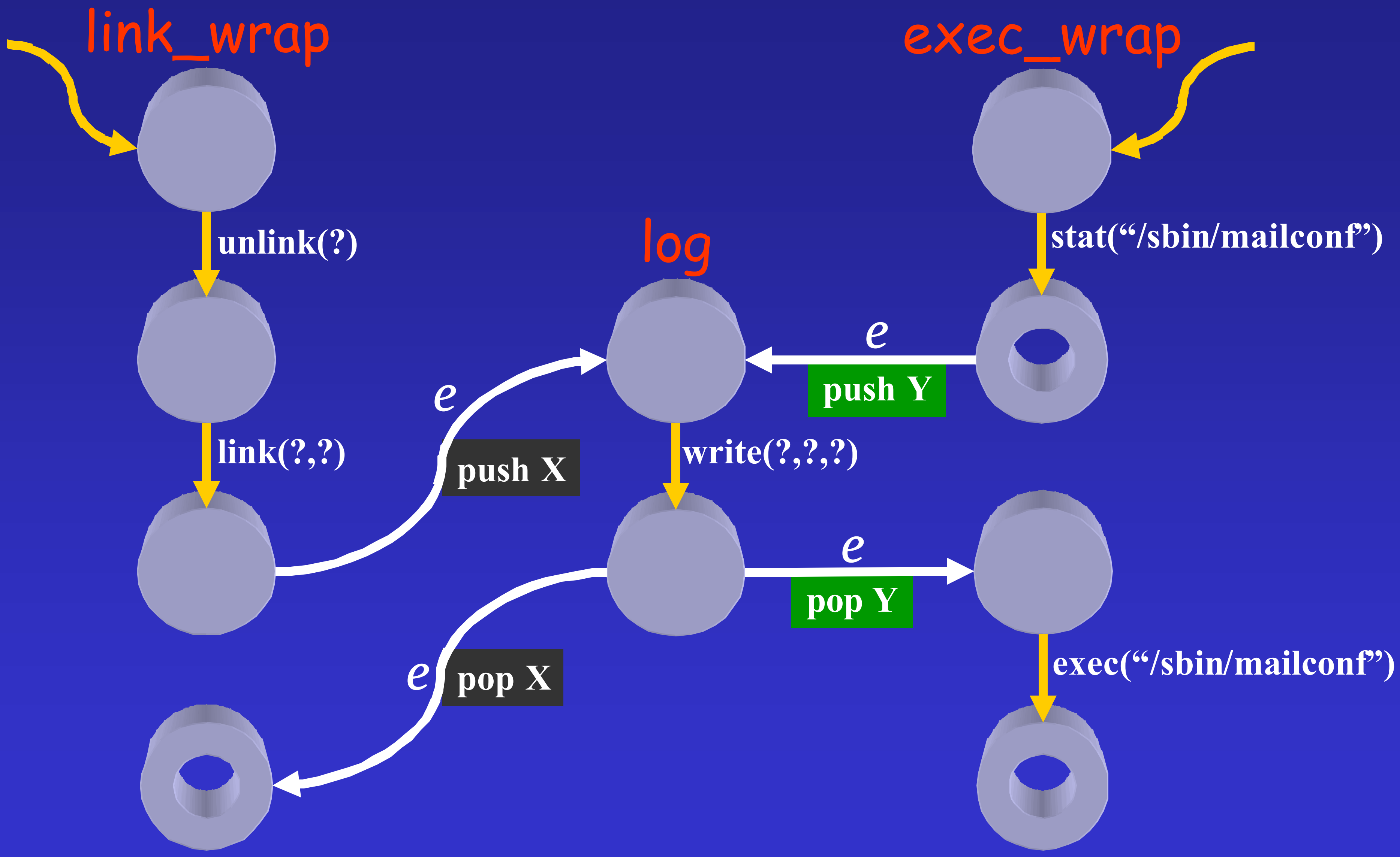
link_wrap
exec_wrap
log
unlink(?)
link(?,?)
log
stat("/sbin/mailconf")
write(?,?,?)
log
exec("/sbin/mailconf")
e
e
e
e

Impossible Path

```
unlink("/sbin/mailconf");
link("/bin/sh", "/sbin/mailconf");
write(-1, 0, 0);
exec("/sbin/mailconf");
```



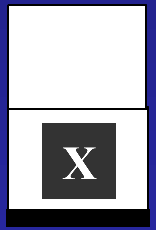
PDA Model



link_wrap
exec_wrap
unlink(?)
log
stat("/sbin/mailconf")
e
push Y
e
link(?,?)
push X
write(?,?,?)
e
pop Y
e
pop X
exec("/sbin/mailconf")

PDA State Explosion

- e-edge identifiers maintained on a stack
  - Stack non-determinism is expensive
  - Unbounded stacks add complexity
  - Best-known algorithm: cubic in automaton size



- Unusable as program model
  - Orders of magnitude slowing of application
    - [Wagner et al. 01, Giffin et al. 02]
  - Conclusion: only weaker NFA models have reasonable performance

Dyck Model

- Efficiently tracks calling context
- As powerful as full PDA
- Efficiency approaches NFA model
- Implication: accuracy & performance can coexist
  - Invalidates previous conclusion

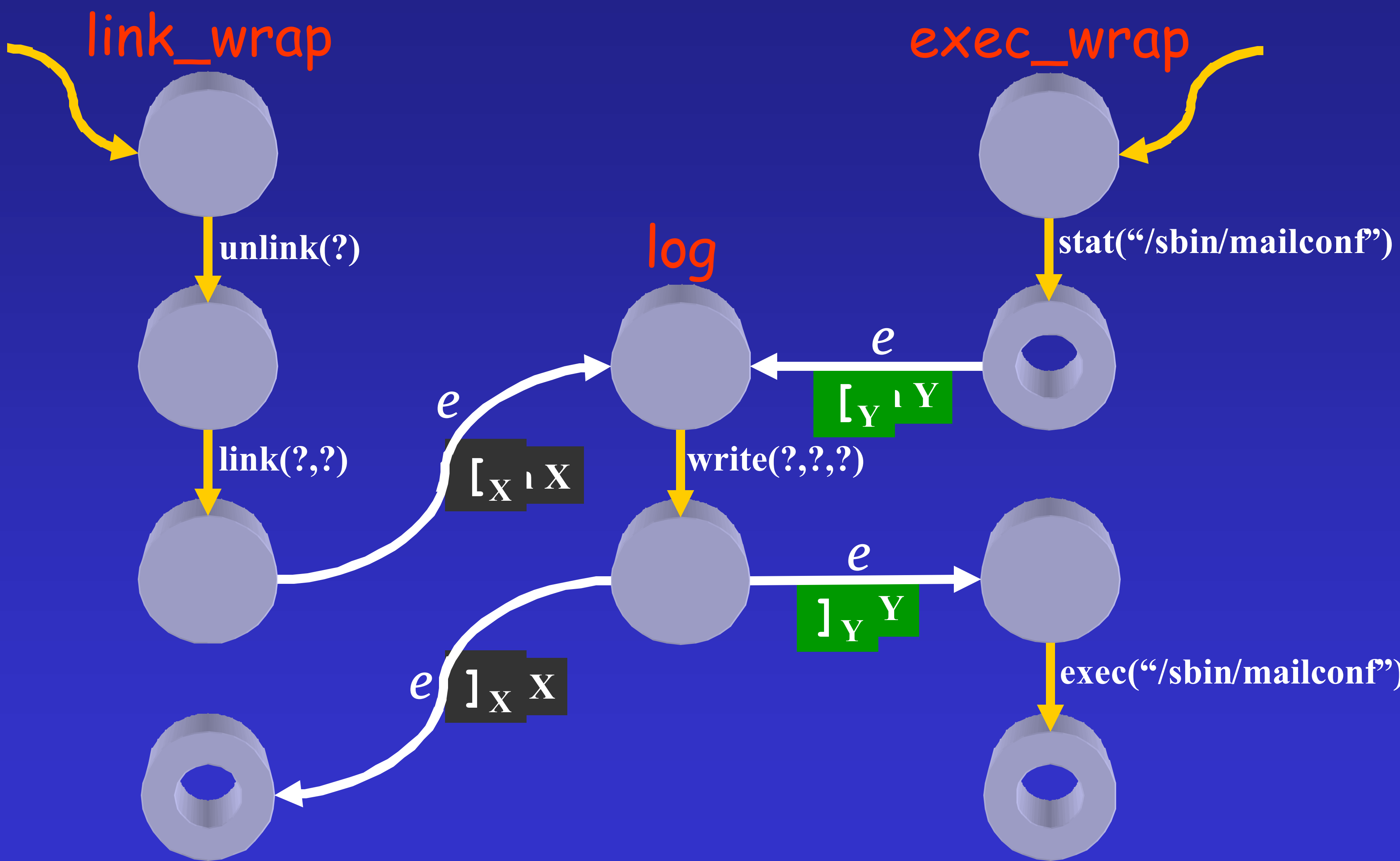
Dyck Model

- Bracketed context-free language
  - [Ginsberg & Harrison 67]

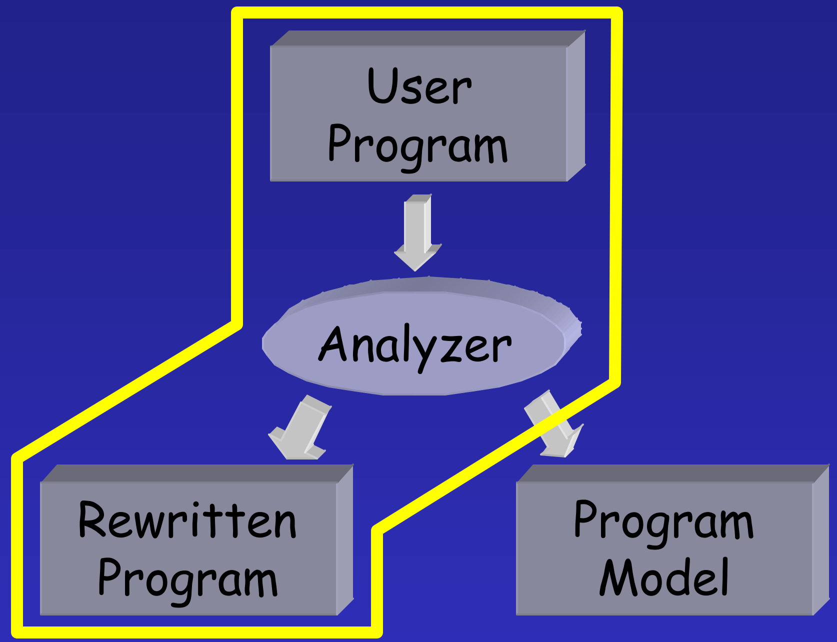
stat $[_y$ write $]_y$ exec
unlink link $[_X$ write $]_X$

- Matching brackets are alphabet symbols
  - Exposes stack operations to runtime monitor
  - Rewrite binary to generate bracket symbols

Dyck Model


link_wrap
exec_wrap
log
unlink(?)
link(?,?)
write(?,?,?)
stat("/sbin/mailconf")
exec("/sbin/mailconf")
e
[X X
e
]X X
e
[Y Y
e
]Y Y

Binary Rewriting



Binary Program → Rewritten Binary

Dyck Null Call Insertion

- Insert code to generate bracket symbols around function call sites
- Notify monitor of stack activity
- Null call squelching prevents high cost

```
void
link_wrap(char *f, char *t)
{
  char msg[BUFFSIZE];

  unlink(t);
  link(f, t);
  snprintf(msg, BUFFSIZE,
    “Linked %s to %s, f, t);
  leftX();
  log(msg);
  rightX();
}
```

Test Programs

Program	Number of Instructions
procmail	112,951
gzip	56,710
eject	70,177
fdformat	67,874
cat	52,028

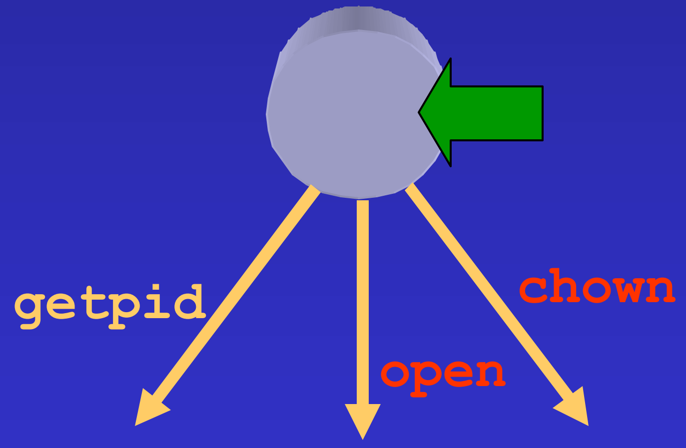
Runtime Overheads

Execution times in seconds

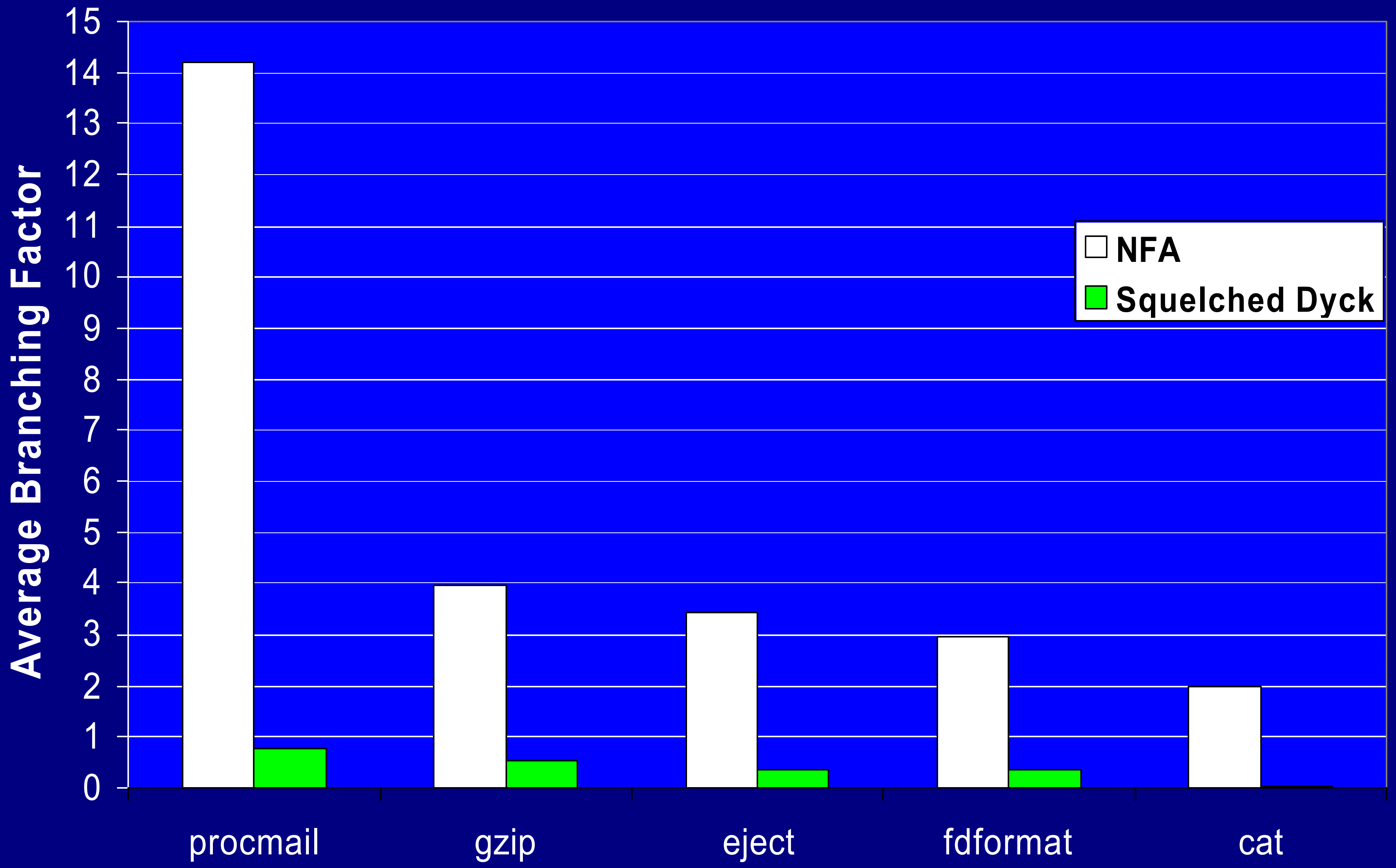
Program	Base	NFA	Increase	Dyck	Increase
procmail	0.42	0.37	0%	0.40	0%
gzip	7.02	6.61	0%	7.16	2%
eject	5.14	5.17	1%	5.22	2%
fdformat	112.41	112.36	0%	112.38	0%
cat	54.65	56.32	3%	80.78	48%

Accuracy Metric

- Average branching factor



NFA and Dyck Model Accuracy

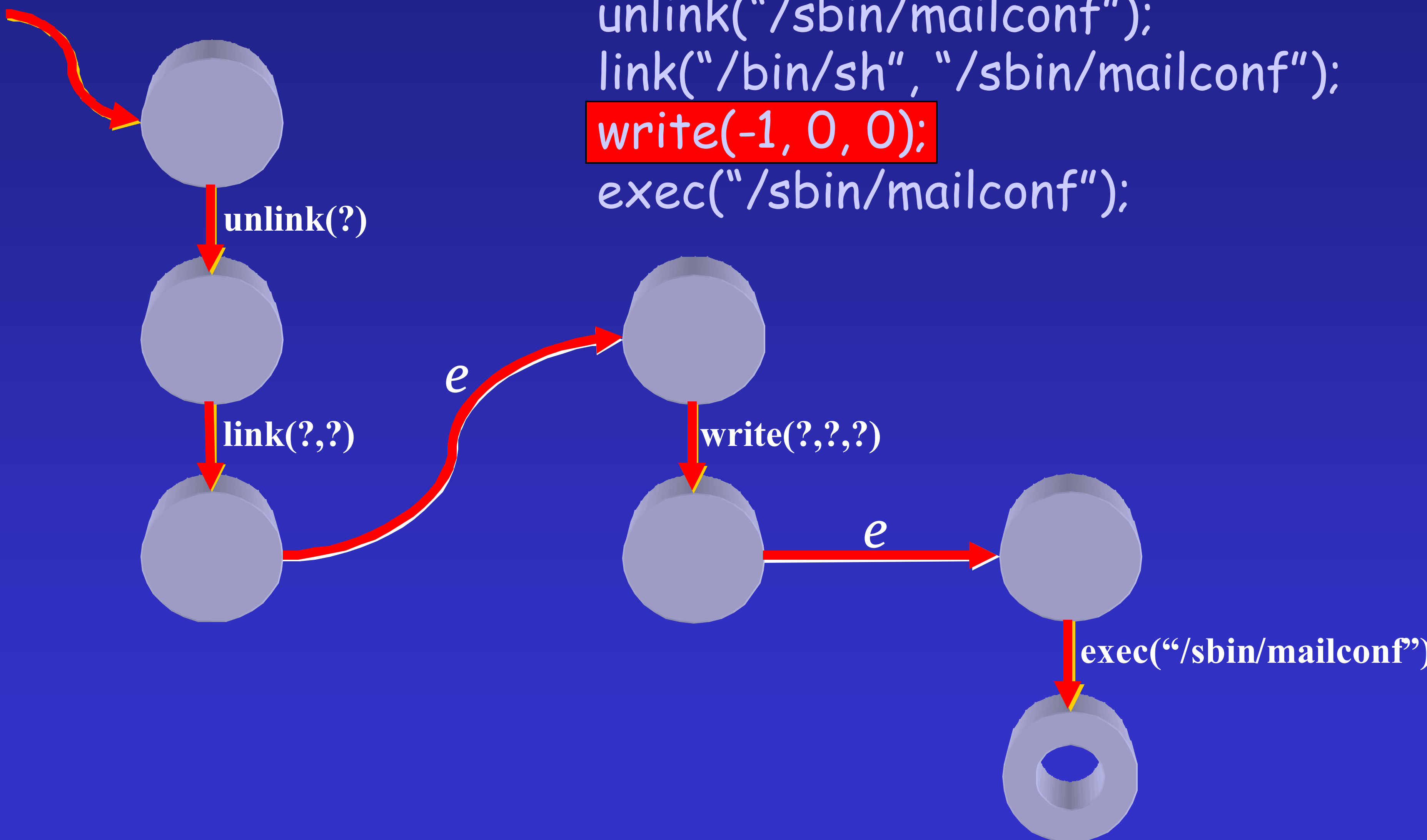


Data-Flow Analysis

- Can use knowledge of argument values to make model more precise.
- Use data-flow analysis of arguments:
  - Argument recovery
    - Sets of constant values
    - Sets of regular expression strings
  - System call return values that control branching
  - Argument dependencies upon system call return values

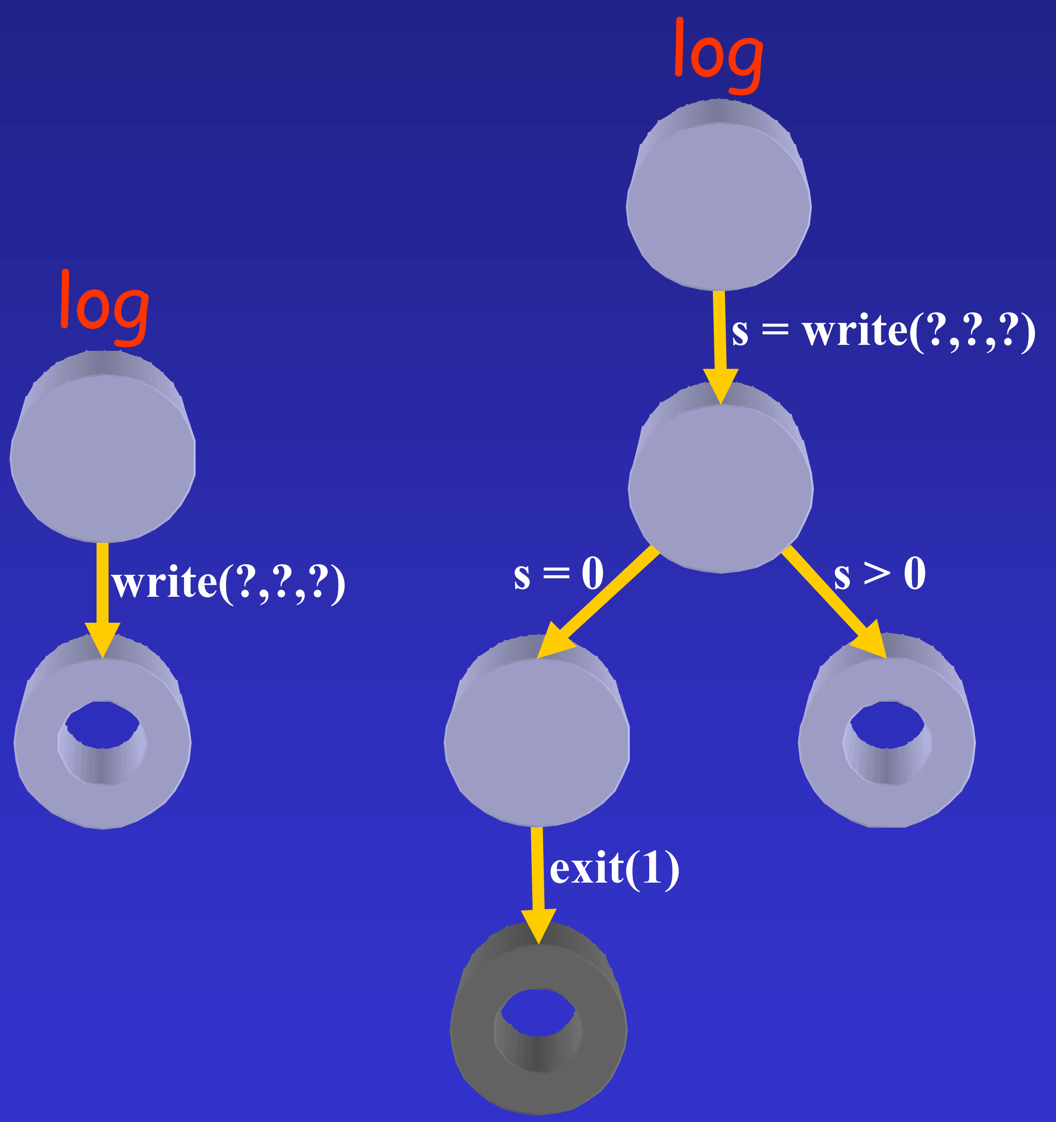
Data-Flow Analysis

```
unlink("/sbin/mailconf");
link("/bin/sh", "/sbin/mailconf");
write(-1, 0, 0);
exec("/sbin/mailconf");
```



Return Value Analysis

```
int log_fd;
void log(const char *m)
{
  int s=strlen(m);
  s=write(log_fd,m,s);
  if (s<=0)
    exit(1);
}
```

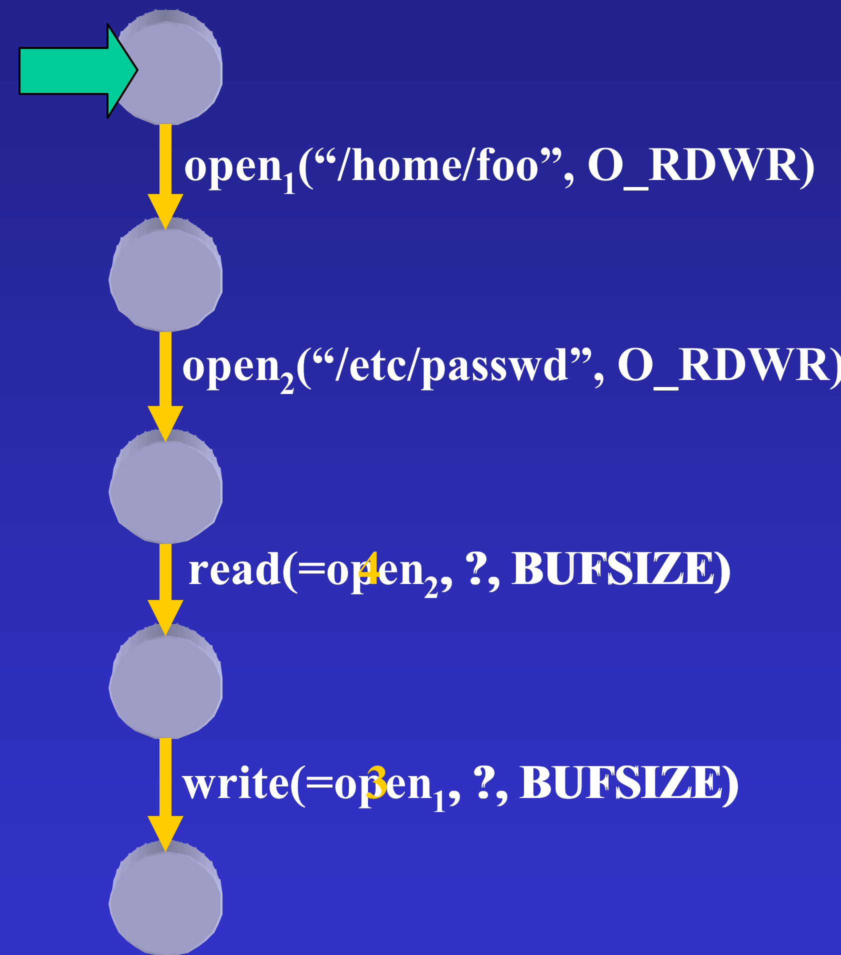


Argument Dependencies

```
. . .
fd1 = open("/home/foo",
           O_RDWR);
fd2 = open("/etc/passwd",
           O_RDWR);
read(fd2, buf, BUFSIZE);
write(fd1, buf, BUFSIZE);
. . .
```

$open_1() = 3;$

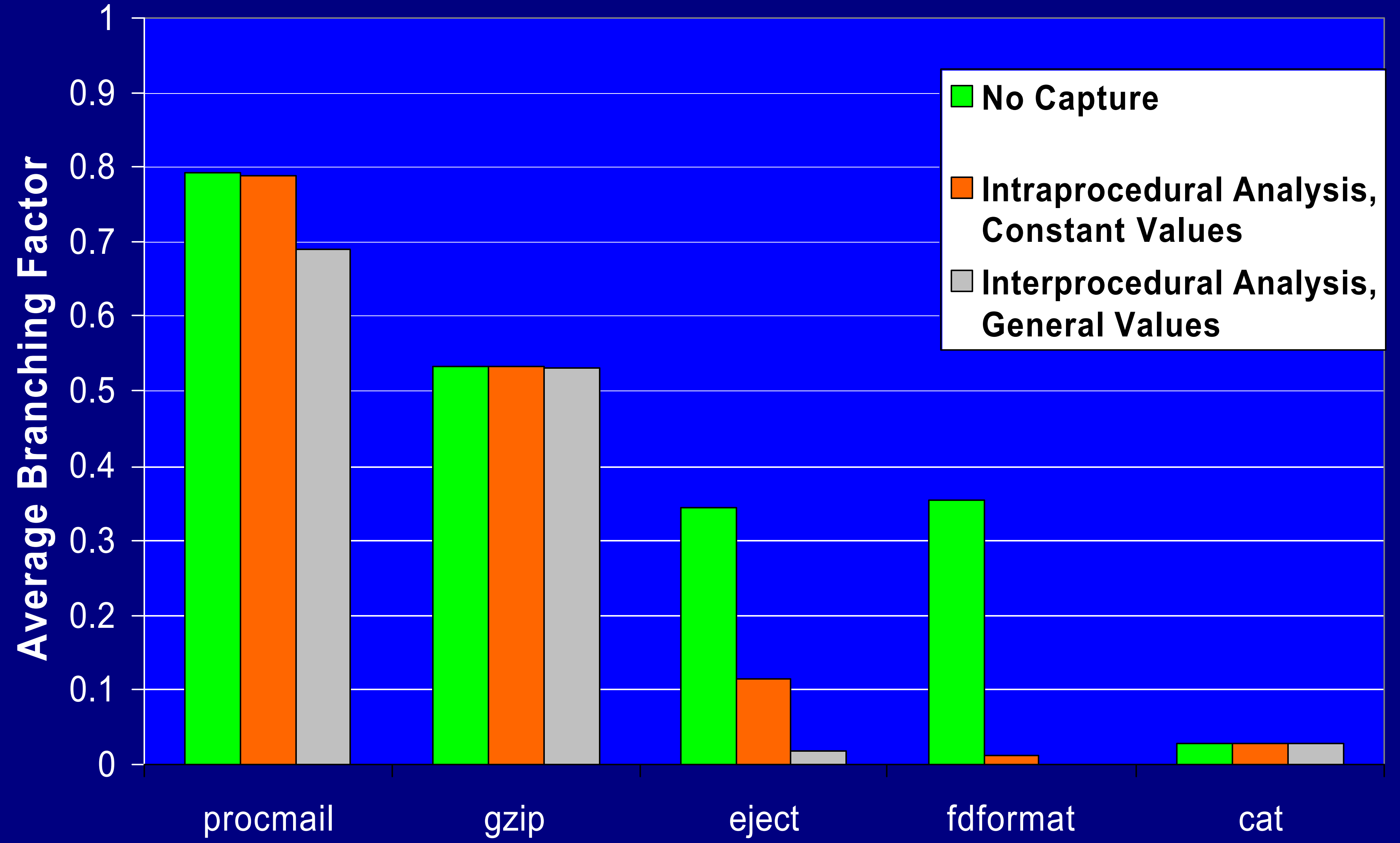
$open_2() = 4;$



Data Flow Sensitivity

Program	Number of System Call Sites	Number Affecting Branches	%
procmail	203	97	48%
gzip	96	54	56%
eject	159	101	64%
fdformat	197	103	52%
cat	108	45	42%

Effects of Argument Capture (Squelched Dyck Model)



Important Ideas

- Model-based intrusion detection forces execution behavior to match model.
- Statically constructed program models historically compromise accuracy for efficiency.
- The Dyck model is the first efficient context-sensitive specification.
- Data-flow analysis restricts undetected attacks by improving model precision.

Efficient Context-Sensitive Intrusion Detection

Jonathon T. Giffin
Somesh Jha, Barton P. Miller
University of Wisconsin
{giffin,jha,bart}@cs.wisc.edu

WiSA - Wisconsin Safety Analyzer
http://www.cs.wisc.edu/wisa