Specification-Based Monitoring

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

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

Problem

- Intrusion detection: How do you know when a process has been subverted?
  - Host-based intrusion detection
  - Remote intrusion detection

- How do malicious users subvert processes?
  - Host-based
  - Via remote execution

Our Solution

Specification-based monitoring

- Specify constraints upon program behavior
  - Static analysis of binary code
- At run-time, ensure execution does not violate specification
  - Limits execution to correct process behavior

Milestones

- Dyck model
  - Efficient & accurate program specification
  - Unites previous work in null calls with specification structure
  - Strong theoretical foundation
- Data-flow-sensitive analysis
  - Branch predicates based upon system call return values
- Second conference paper (submitted)

Overview

Static binary analysis for intrusion detection

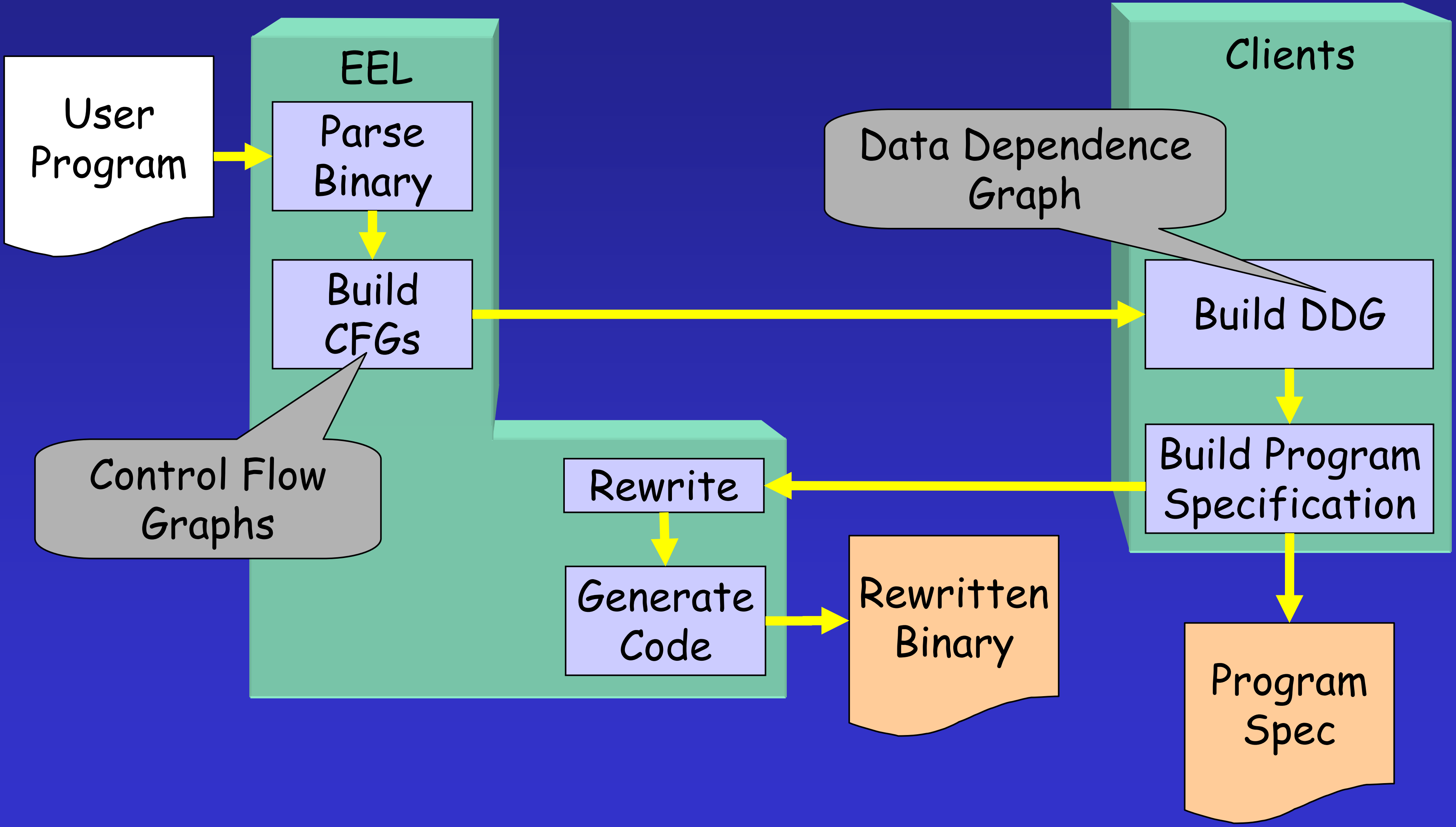
- Shortcomings with prior program models
- Dyck model addresses shortcomings
- Null call squelching
- Measurements of accuracy & efficiency

Specification-Based Monitoring

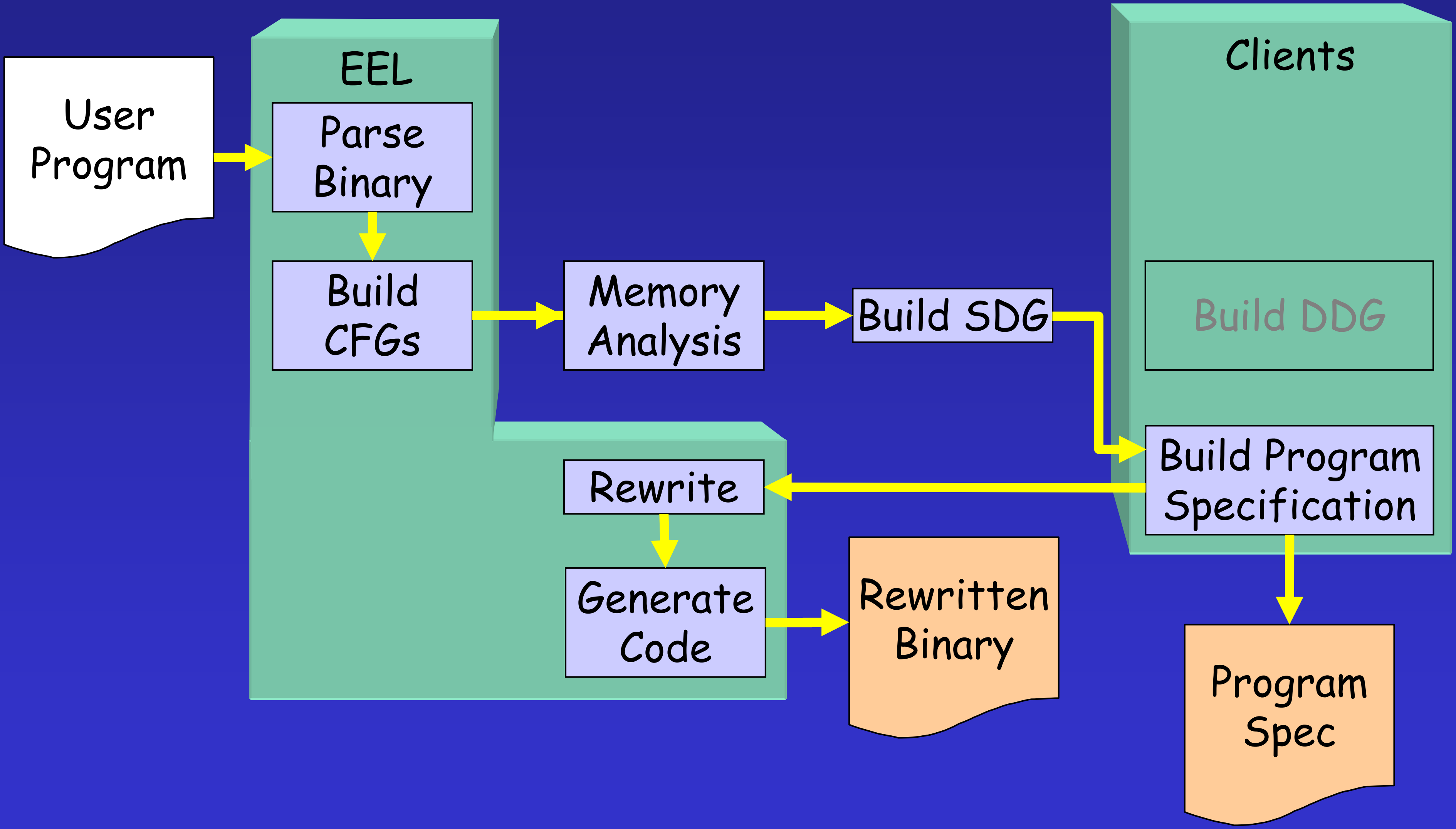
- Specify constraints upon program behavior
  - Static analysis of binary code
  - Construct automaton modeling all system call sequences the program can generate

- Ensure execution does not violate specification
  - Operate the automaton
  - If no valid states, then intrusion attempt occurred

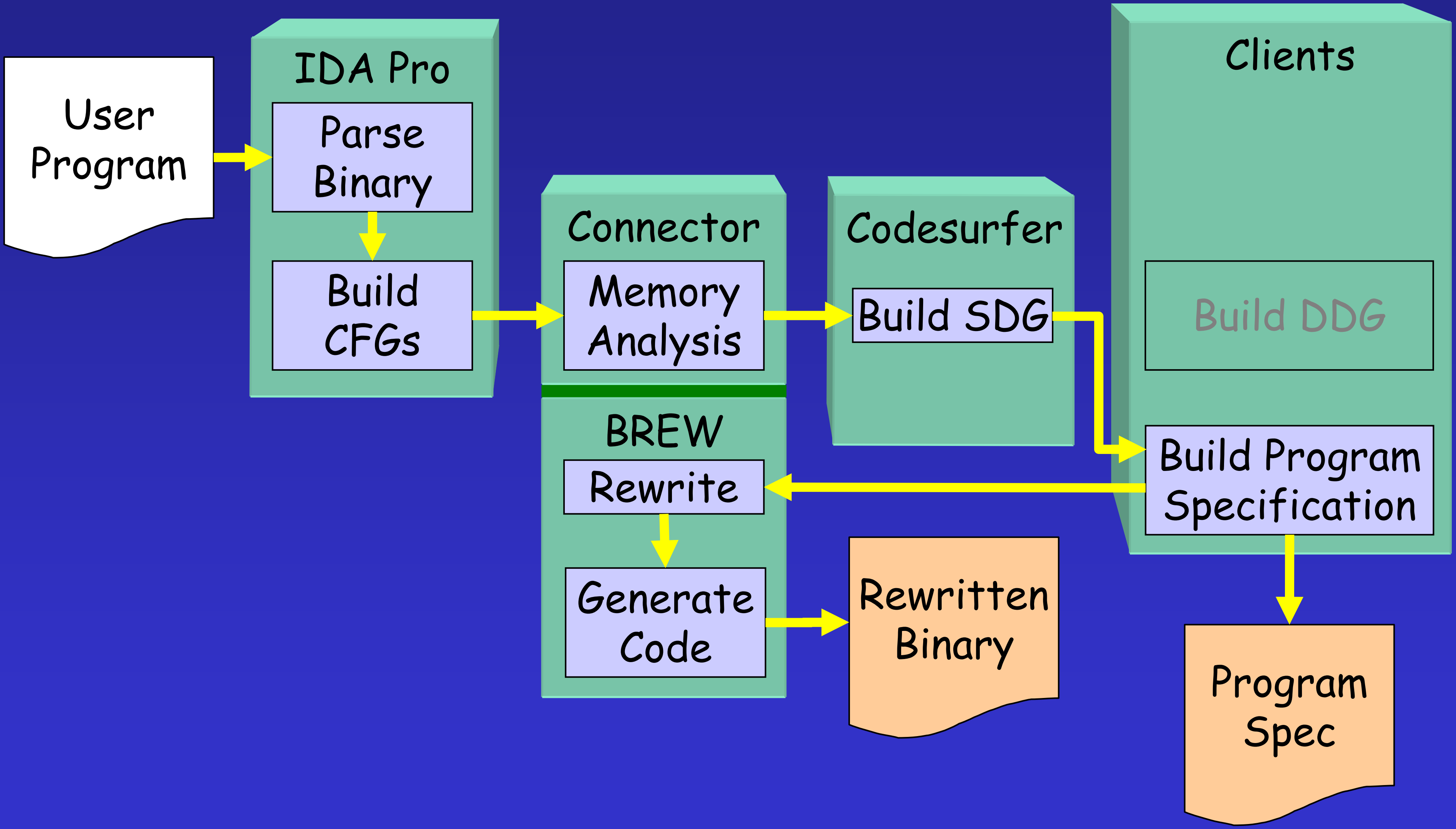
Architecture



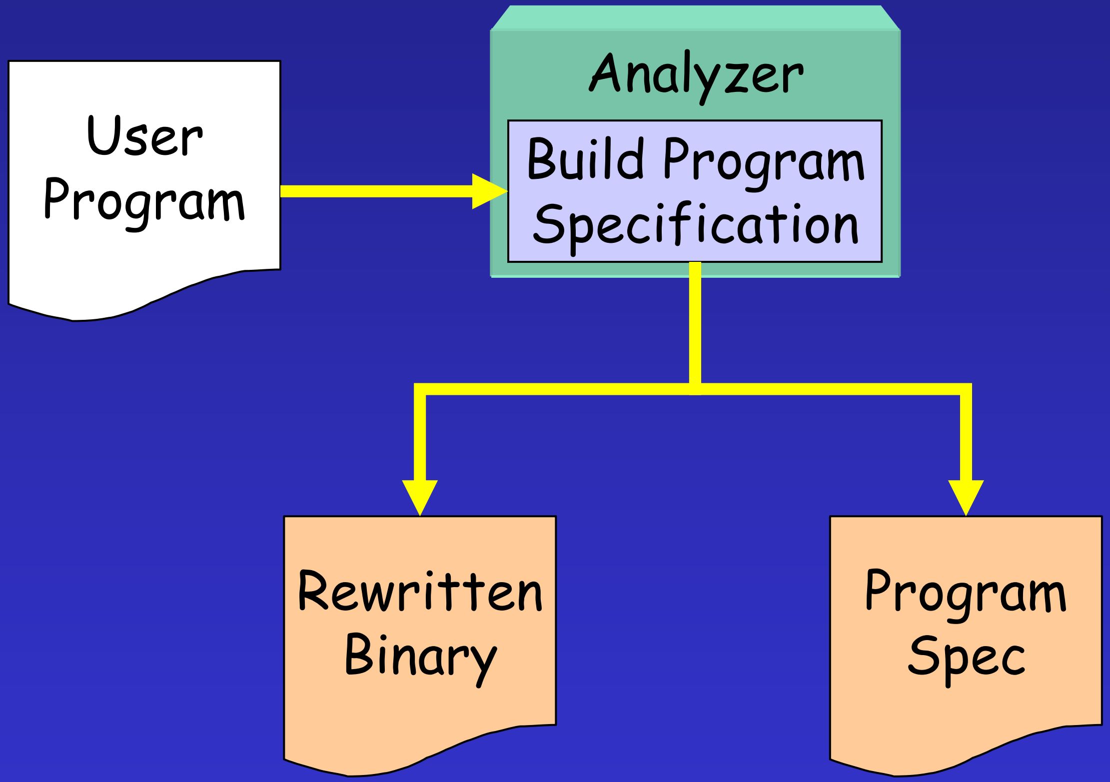
Architecture



Architecture



Simplified Architecture



User Program

Analyzer

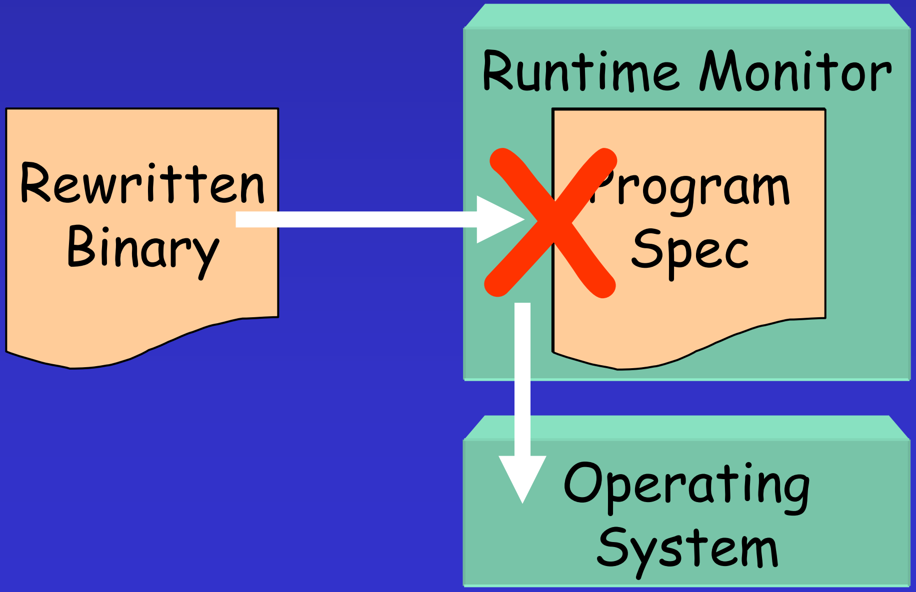
Build Program Specification

Rewritten Binary

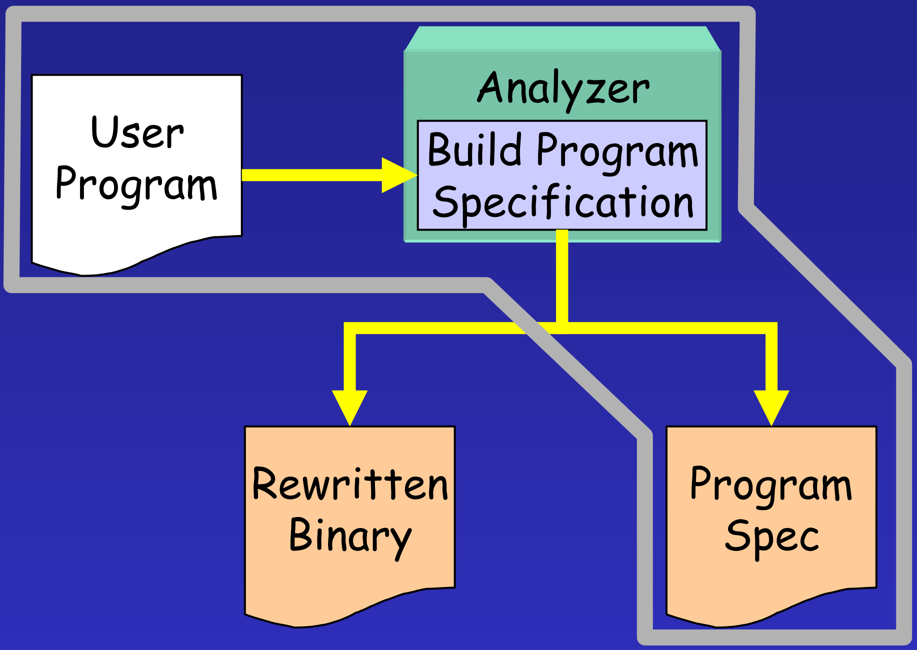
Program Spec

Specification-Based Monitoring

- Our runtime monitor monitors program execution at some event interface layer
  - Operating system kernel traps
  - Remote system calls
- Ensures program events match specification



Model Construction



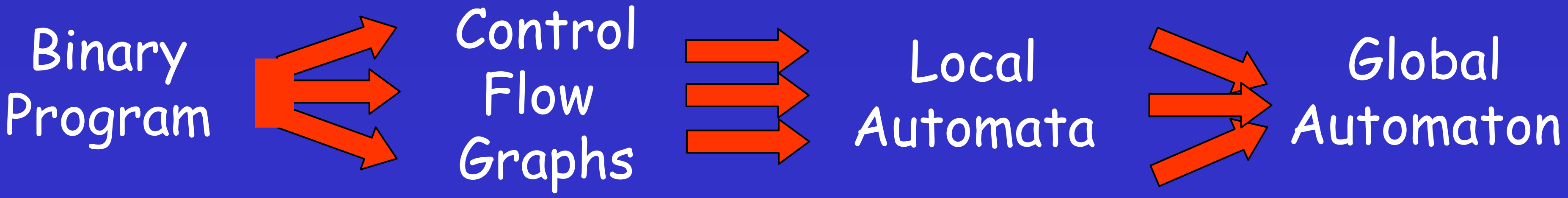
User Program

Analyzer

Build Program Specification

Rewritten Binary

Program Spec



Binary Program

Control Flow Graphs

Local Automata

Global Automaton

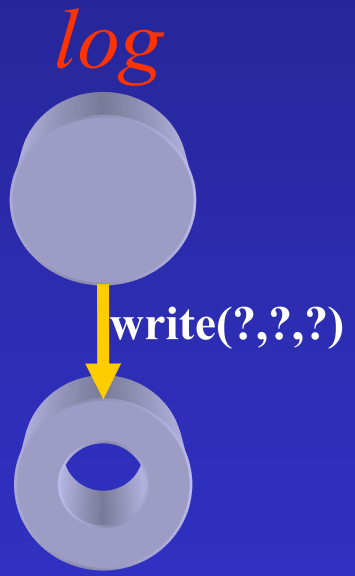
Code Example

```
int log_fd;
void log(const char *m)
{
  int s=strlen(m);
  s=write(log_fd,m,s);
}
```

```
log:
  save %sp, -96, %sp
  call strlen
  mov %i0, %o0
  mov %o0, %o2
  mov %i0, %o1
  call write
  ld [log_fd], %o0
  ret
  restore
```

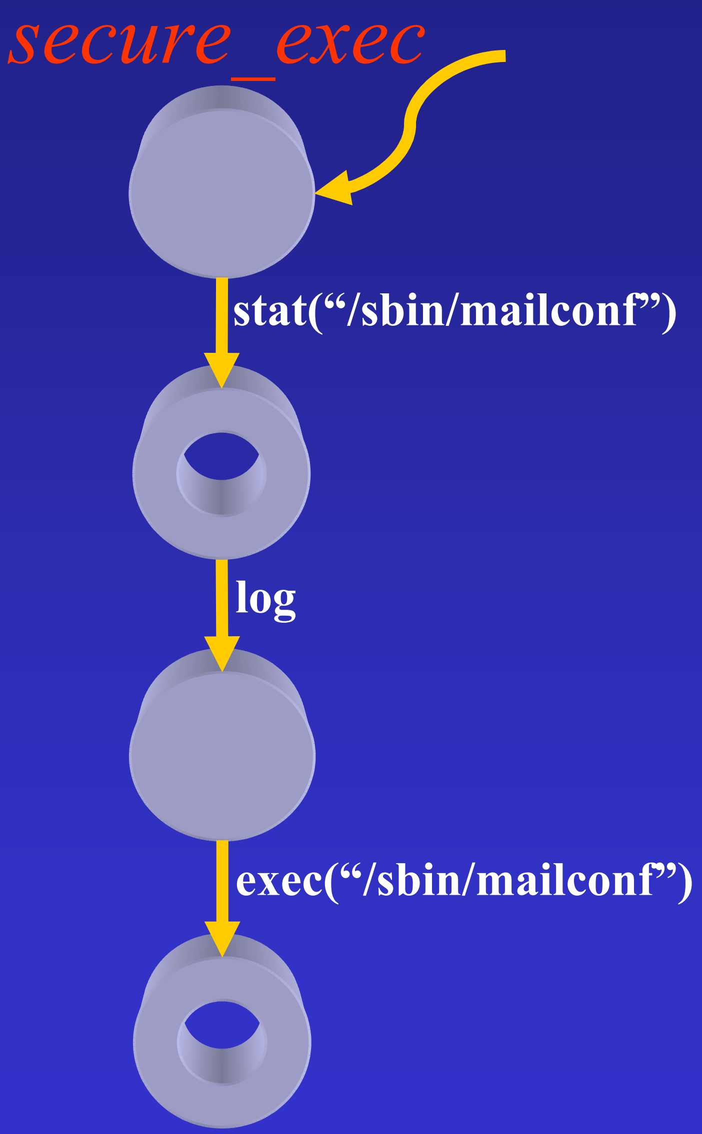
Code Example

```
int log_fd;
void log(const char *m)
{
  int s=strlen(m);
  s=write(log_fd,m,s);
}
```

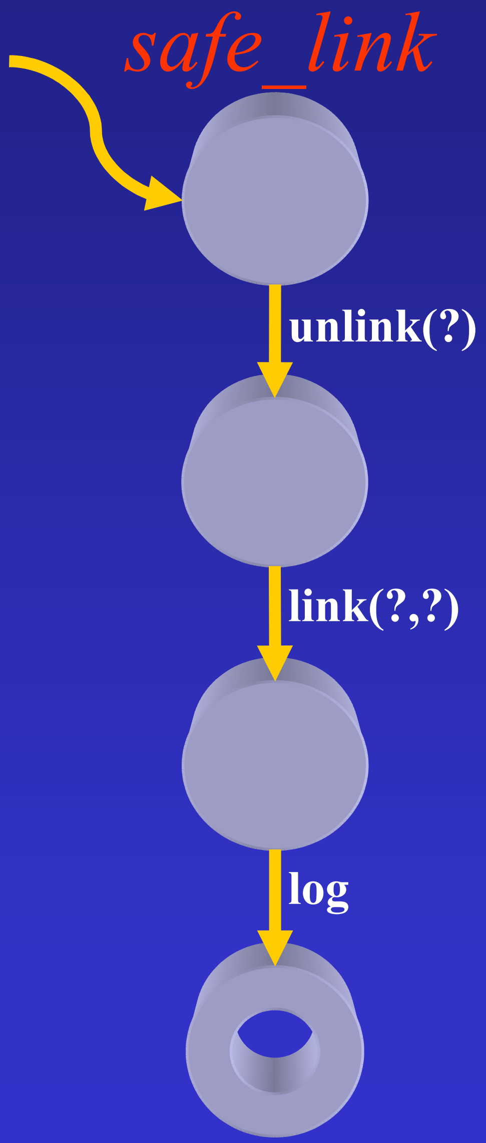


Code Example

```
void secure_exec()
{
  struct stat s;
  stat(“/sbin/mailconf”,&s);
  if (s.st_size == 1013288)
  {
    log(“execing conf”);
    exec(“/sbin/mailconf”);
  }
}
```

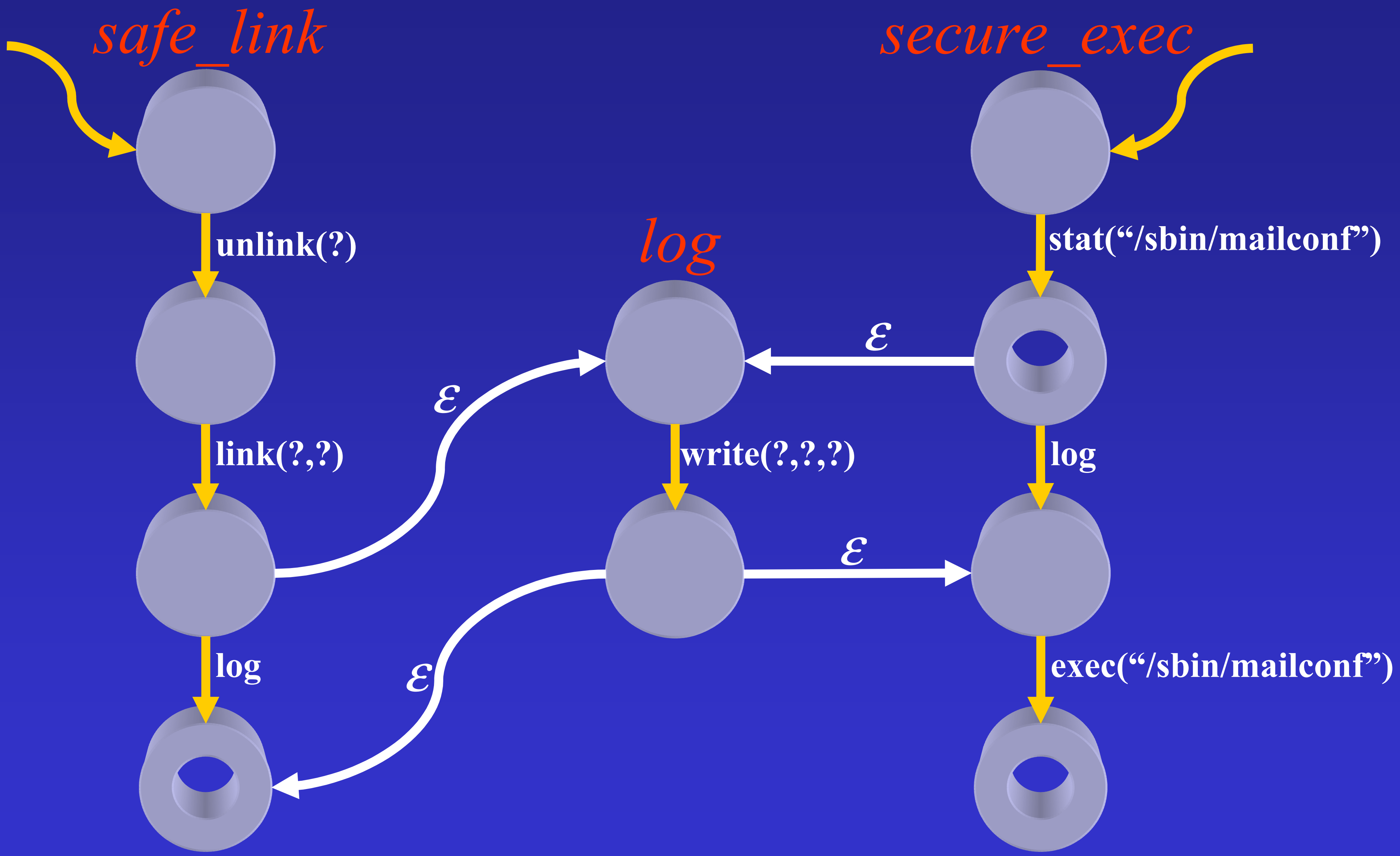


Code Example



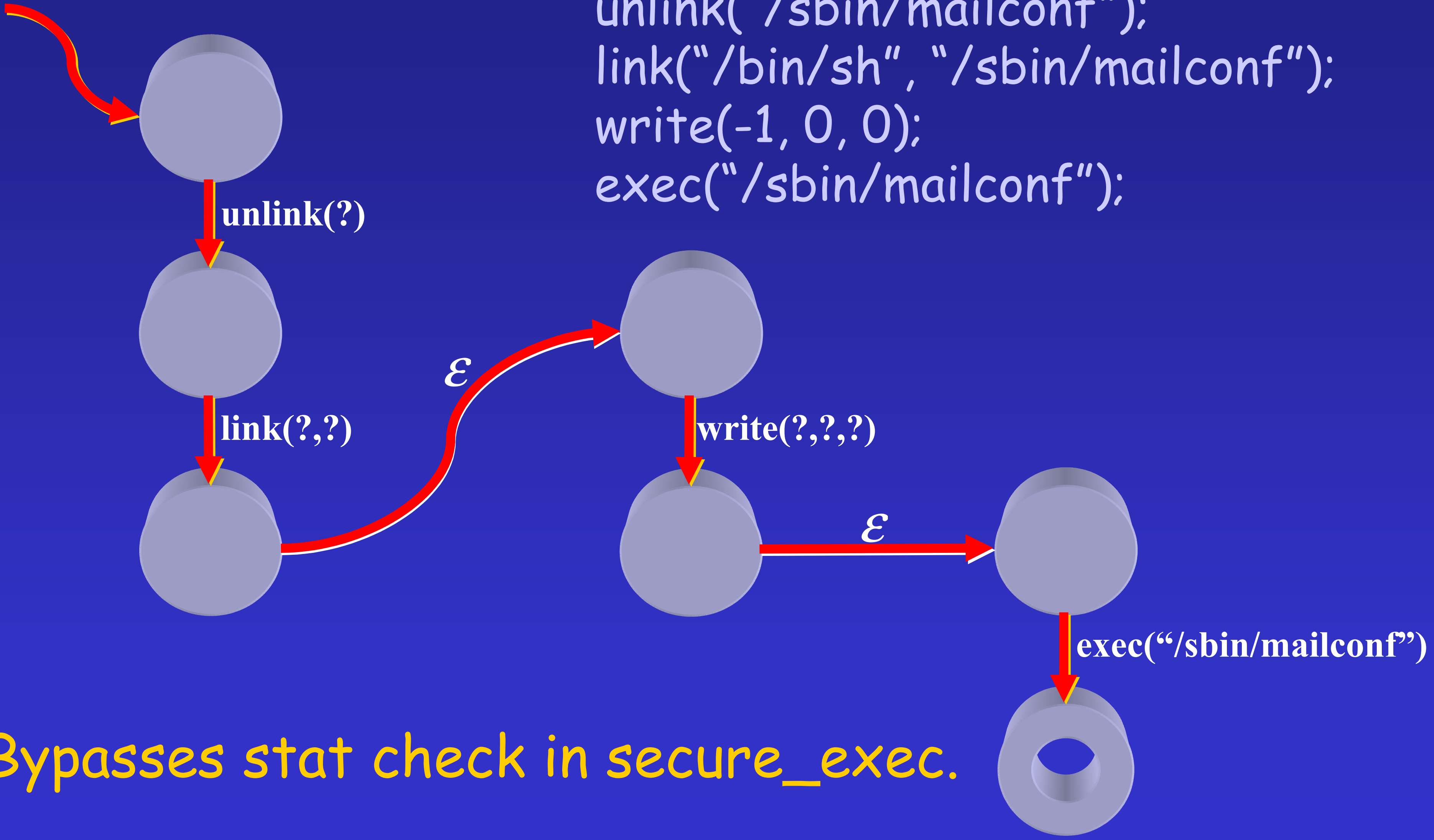
```
void safe_link(char *f, *t)
{
   char msg[500];

   unlink(t);
   link(f, t);
   snprintf(msg, 50,
     “Linked %s to %s, f, t);
   log(msg);
}
```

NFA Model


safe_link
log
secure_exec
unlink(?)
stat("/sbin/mailconf")
ε
ε
link(?,?)
write(?,?,?)
log
ε
ε
log
exec("/sbin/mailconf")

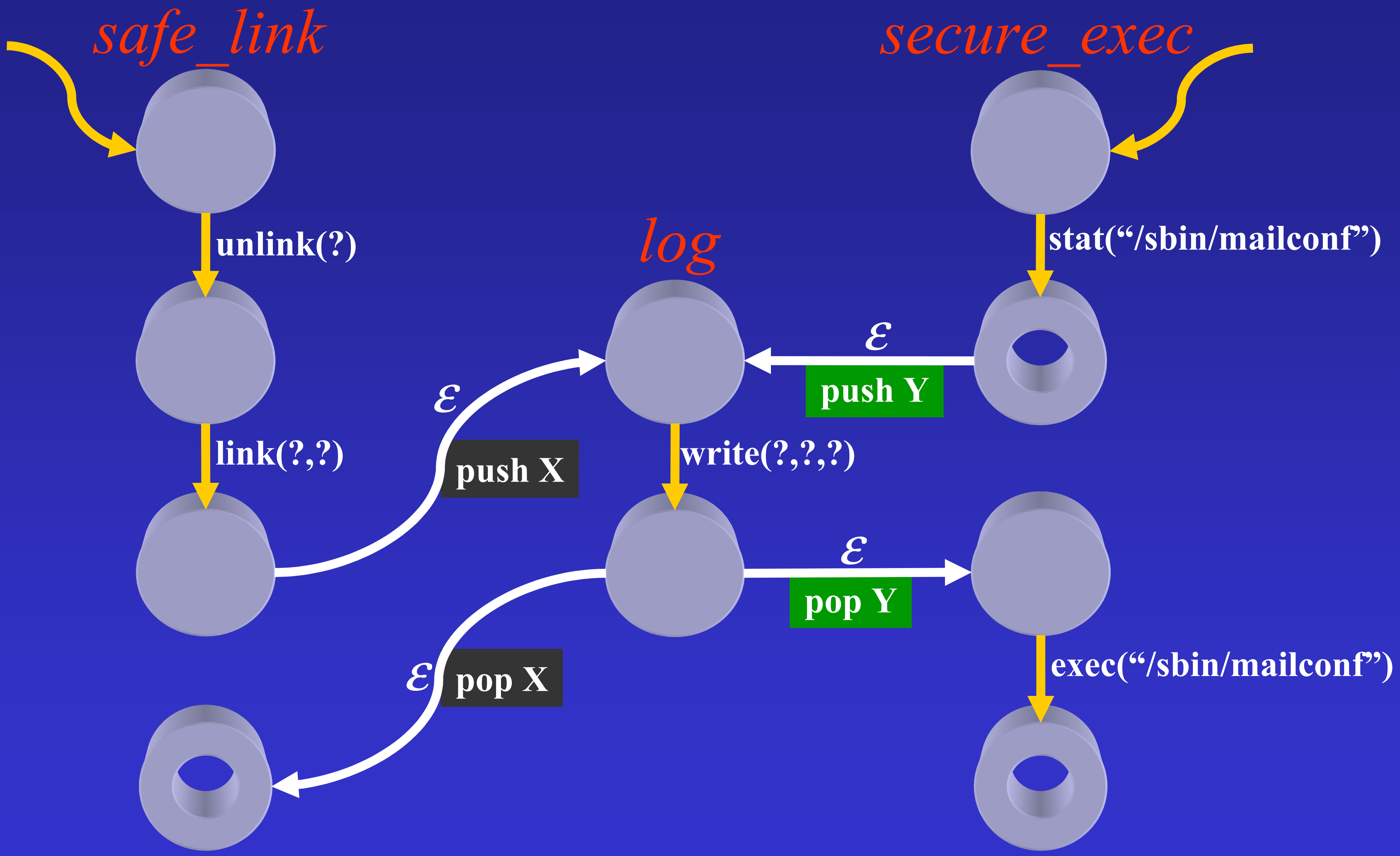
Impossible Paths

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



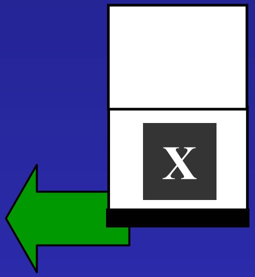
Bypasses stat check in secure_exec.

PDA Model


safe_link
secure_exec
unlink(?)
log
stat("/sbin/mailconf")
ε
push Y
ε
link(?,?)
push X
write(?,?,?)
ε
pop Y
ε
pop X
exec("/sbin/mailconf")

PDA State Explosion

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



- First solution
  - Use stack abstractions: bound stack height
  - Practically limited to size 0 [NFA]

- Null call insertion reduced non-determinism
  - Increased stack bound to 4

Dyck Model

- Efficiently tracks calling context

- As powerful as full PDA
- Efficiency approaches NFA model

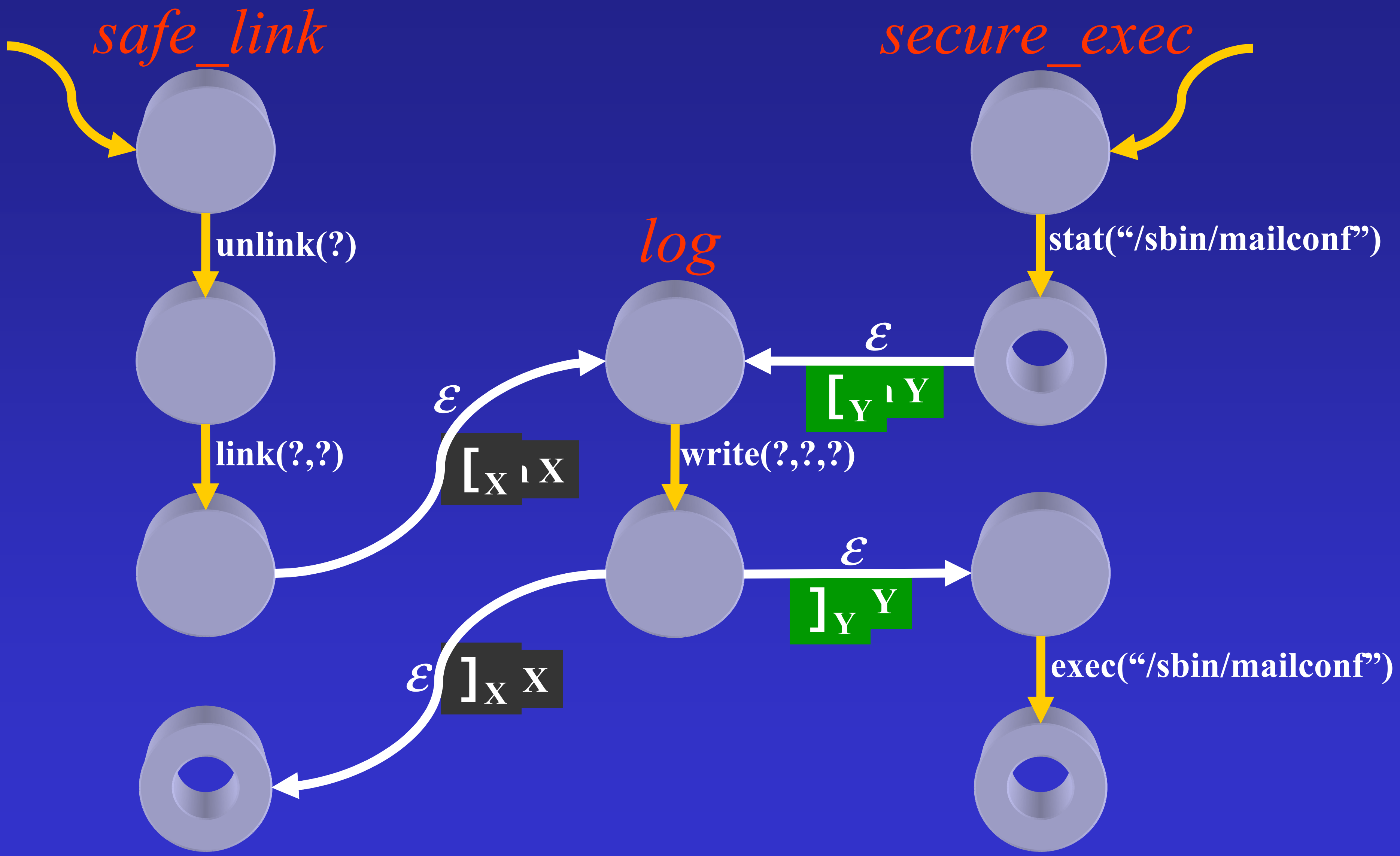
Dyck Model

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

$[_y$ write $]_y$ exec
unlink symlink $[_x$ write $]_x$

- Matching brackets are alphabet symbols
  - Null calls can serve this purpose
  - Exposes stack operations to runtime monitor

Dyck Model


safe_link
secure_exec
log
unlink(?)
link(?,?)
stat("/sbin/mailconf")
ε
$[_Y$ Y
ε
$[_X$ X
write(?,?,?)
ε
$]_Y$ Y
exec("/sbin/mailconf")
ε
$]_X$ X

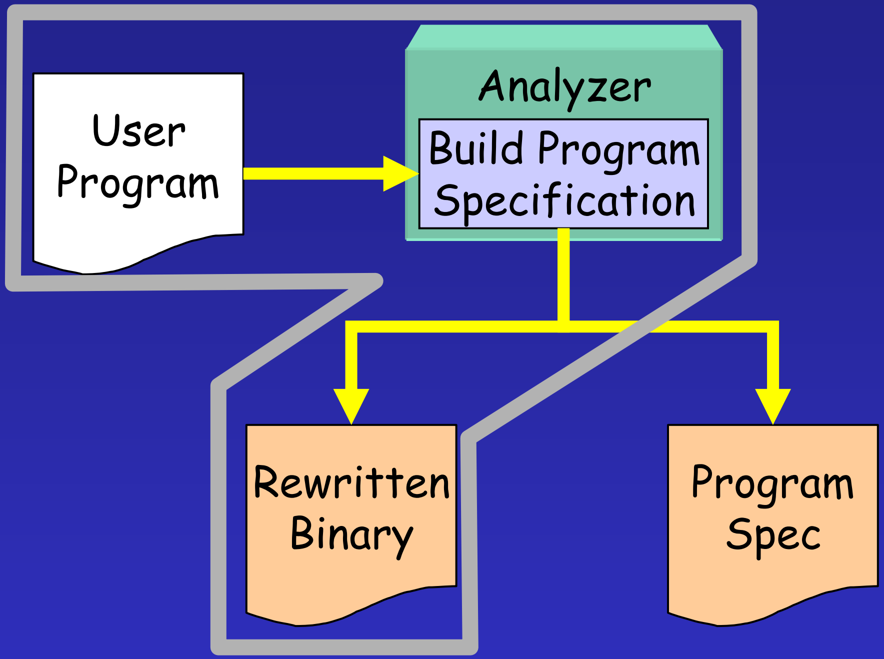
Dyck Model

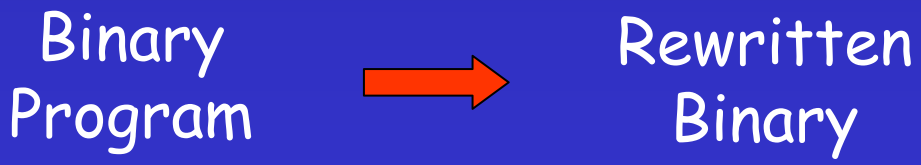
- Subsequences of bracket symbols form a Dyck language
[Chomsky & Scheutzenberger 63]
[von Dyck 1882]

- Dyck languages maintain PDA expressiveness
[Chomsky 62]

$$L = h(D \cap R)$$

Binary Rewriting





Binary Rewriting

```
void secure_exec()
{
  struct stat s;
  stat(“/sbin/mailconf”,&s);
  if (s.st_size == 1013288)
  {
    leftX();
    log(“execing conf”);
    rightX();
    exec(“/sbin/mailconf”);
  }
}
```

- Insert dummy system calls around function call sites
- Notify monitor of stack activity

Null Call Squelching

- Naïve Dyck call insertion leads to high cost
- Null call squelching eliminates irrelevant null calls

Relevant: … $[_x$ `write` $]_x$ …

Irrelevant: … $[_x$ $]_x$ …

Null Call Squelching

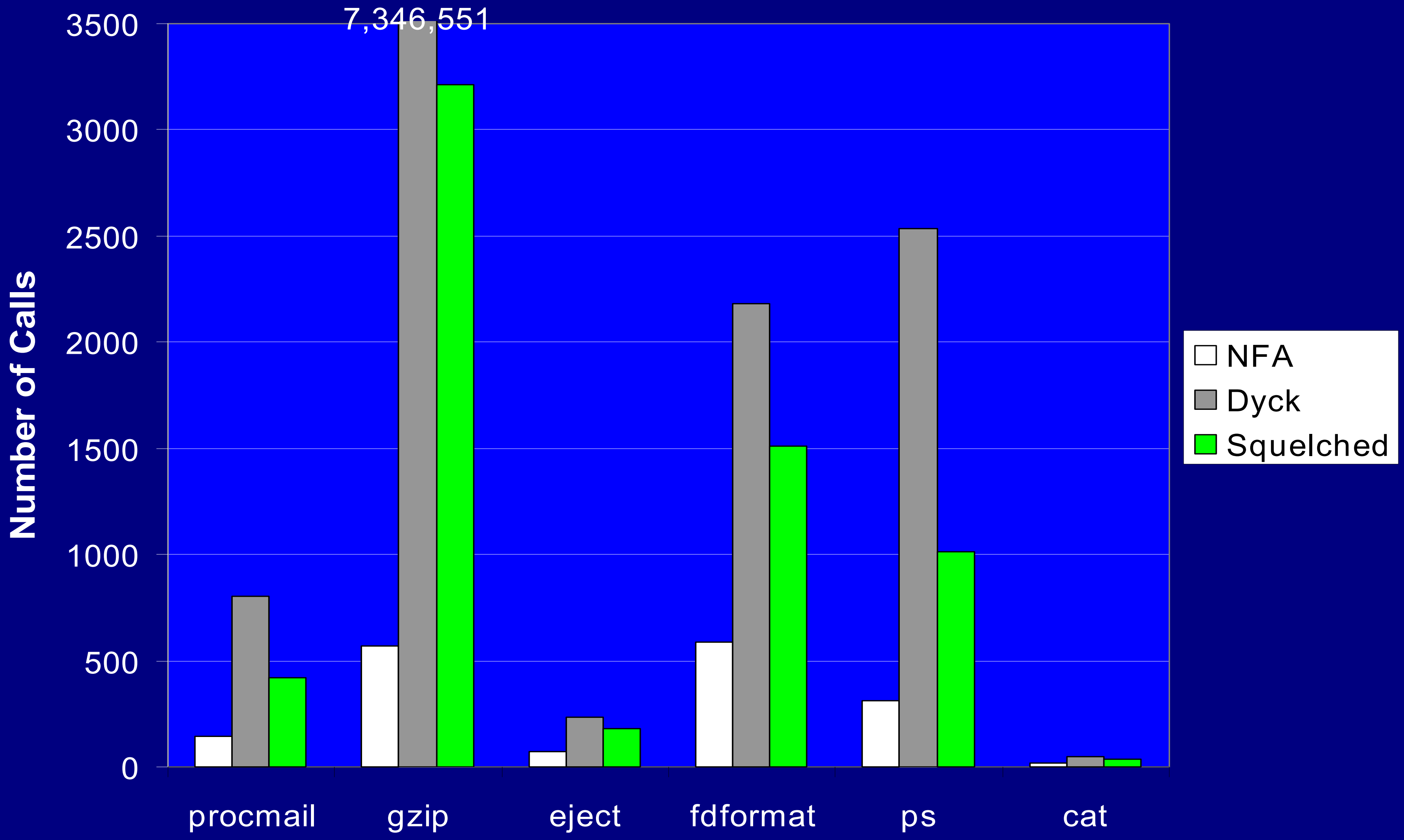
Upper bound:

- The squelched Dyck model generates at most $2hn$ null calls
  - h is the diameter of the call graph
  - n is the number of true system calls executed

Test Programs

Program	Number of Instructions
procmail	107,246
gzip	56,710
eject	70,177
fdformat	67,874
ps	59,814
cat	54,028

Number of Monitor Calls Generated



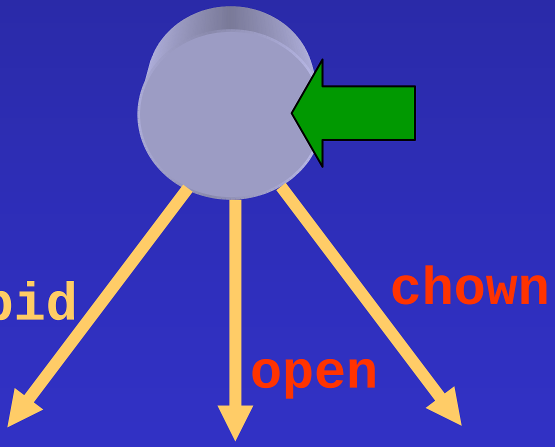
Runtime Overheads

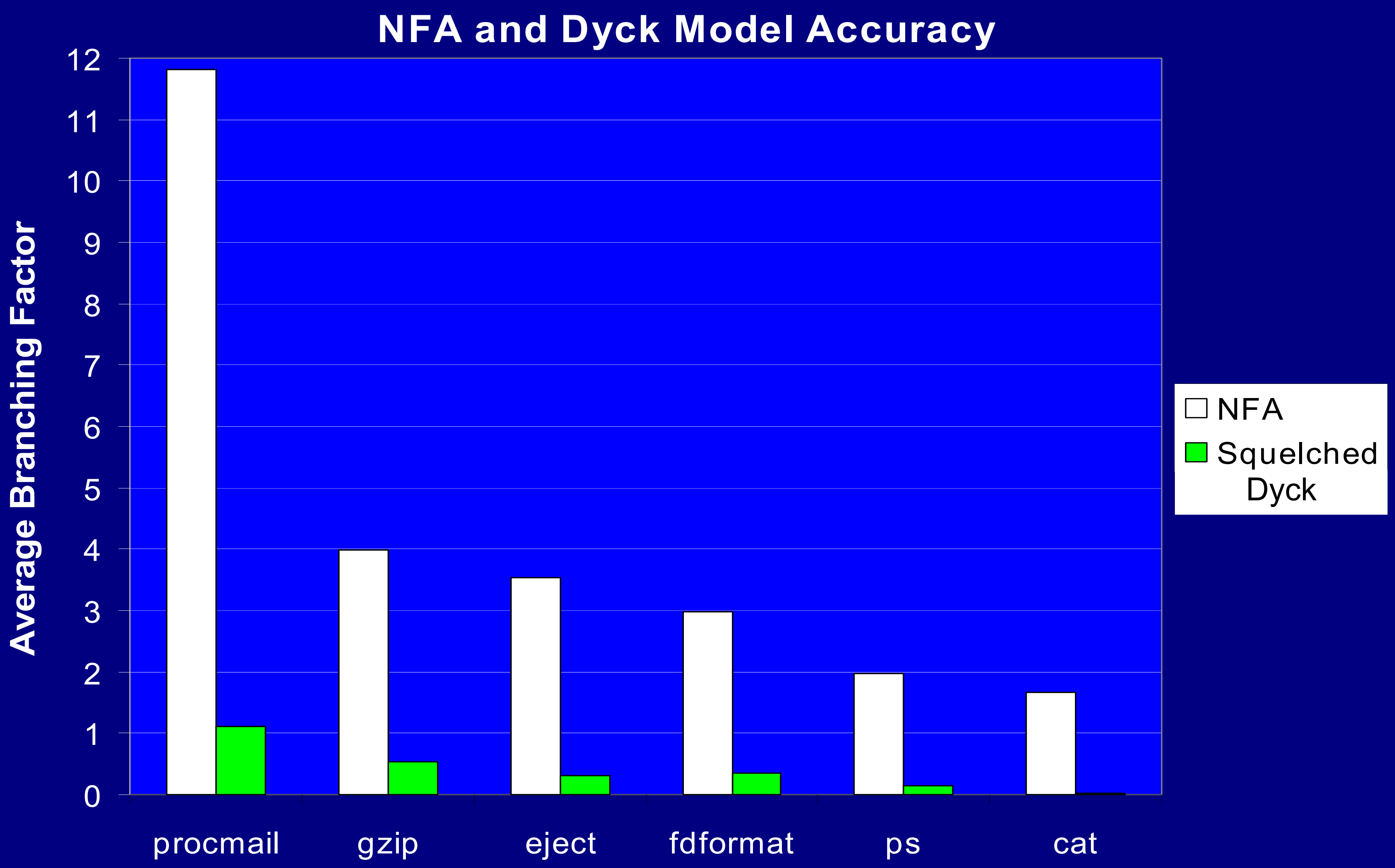
Program	NFA	Dyck			
		Original	Increase	Squelched	Increase
procmail	0.27	0.70	160 %	0.21	---
gzip	7.36	1381.33	18700 %	8.80	19.6%
eject	5.67	5.44	---	5.35	---
fdformat	112.01	112.38	0.3%	112.42	0.4%
ps	0.36	0.94	160 %	0.31	---
cat	0.00	0.00	---	0.00	---

Execution times in seconds

Accuracy Metric

- Average branching factor




NFA and Dyck Model Accuracy
Average Branching Factor
0
1
2
3
4
5
6
7
8
9
10
11
12
procmail
gzip
eject
fdformat
ps
cat
☐ NFA
🟩 Squelched Dyck

Important Ideas

- Statically constructed program models historically compromise accuracy for efficiency.
- The Dyck Model is the first efficient context-sensitive specification.
- Null call squelching bounds the number of Dyck null calls generated.

Specification-Based Monitoring

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

WiSA - Wisconsin Safety Analyzer