Efficient Context-Sensitive Intrusion Detection

Jonathon T. Giffin
Somesh Jha
Barton P. Miller

University of Wisconsin

{giffin,jha,bart}@cs.wisc.edu

Intrusion Detection Problems

- Detection ability: How do you know when a process has been subverted?
  - Host-based intrusion detection
  - Remote intrusion detection
- Detection efficiency: Can subversion be detected in real-time?

Our Solution

Model-based anomaly detection

- 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
  - Strong theoretical foundation
  - Demonstrated that program state exposure improves performance
  - Two new papers published
    - [NDSS 2004, Oakland 2004]
- Static analysis infrastructure
  - Analyze dynamically-linked executables
  - Build Dyck models without binary rewriting

Overview

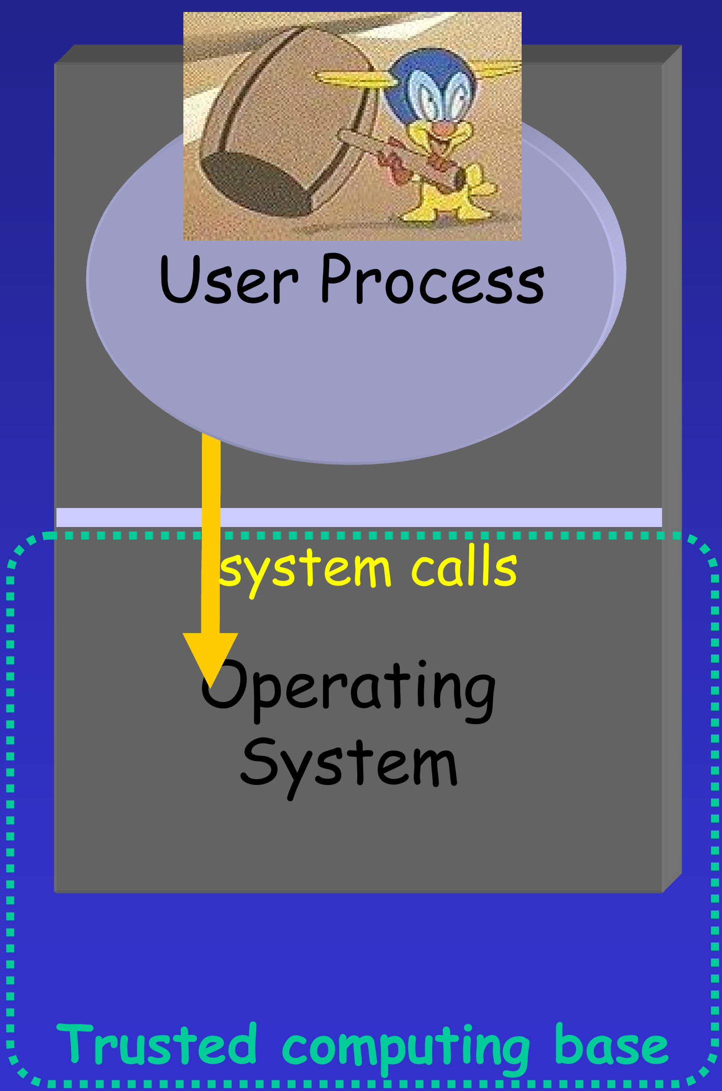
Attacks

- Server attack (conventional host-based IDS)
- Remote execution attack (remote IDS)

Model-based intrusion detection

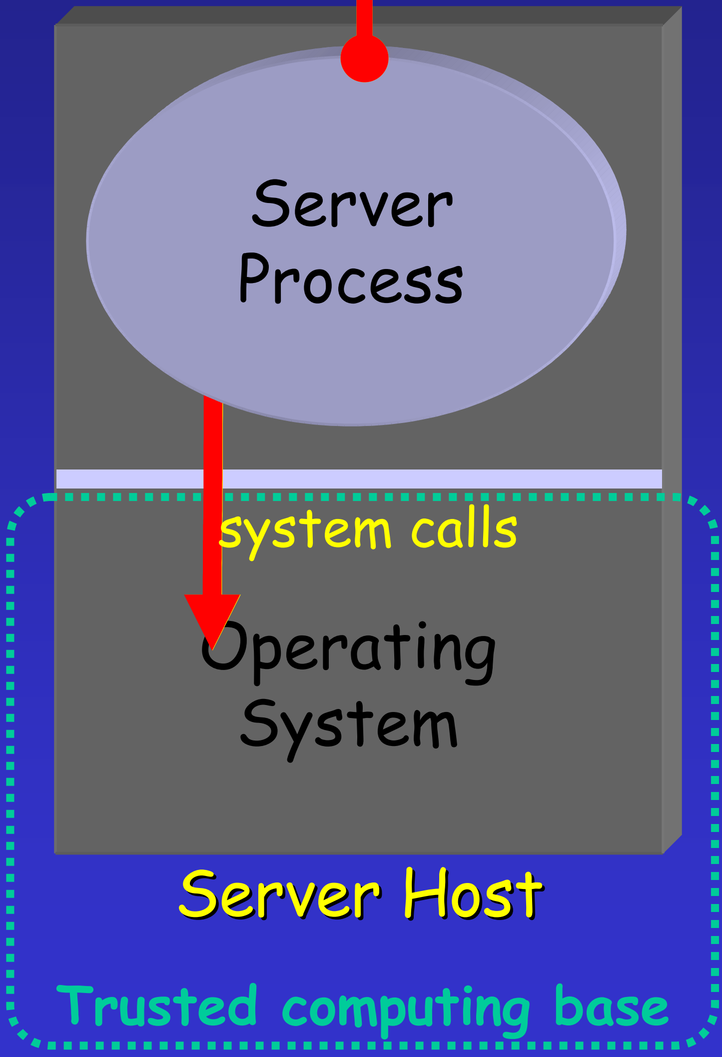
- Constructing program models using static binary analysis
- Accuracy/performance tradeoff in prior models
- Deterministic PDA models solve tradeoff
- Combining static & dynamic analysis

Worldview



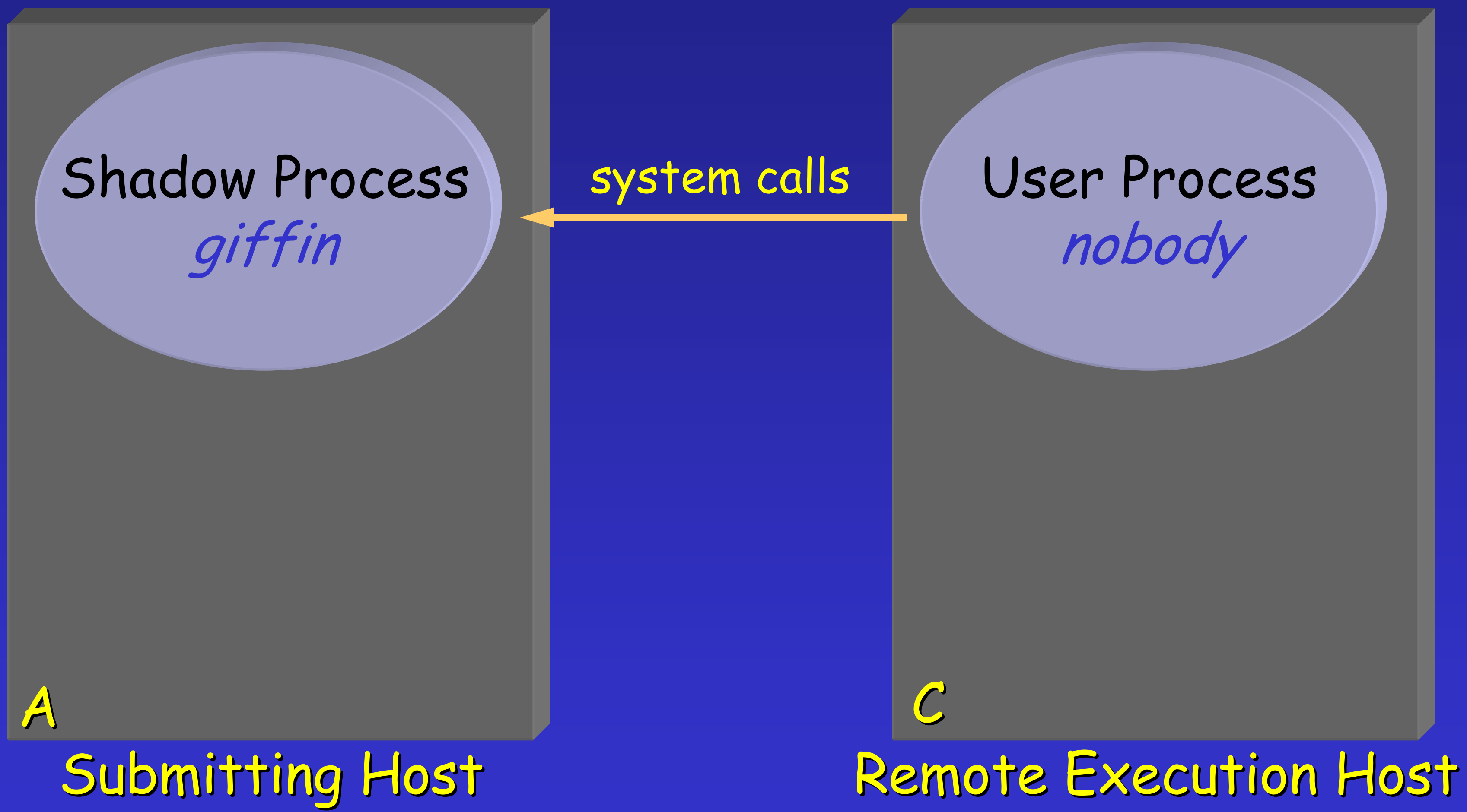
- Running processes make operating system requests
- Changes to trusted computing base done via these requests
- Attacker subverts process to generate malicious requests

Example: Server Attack

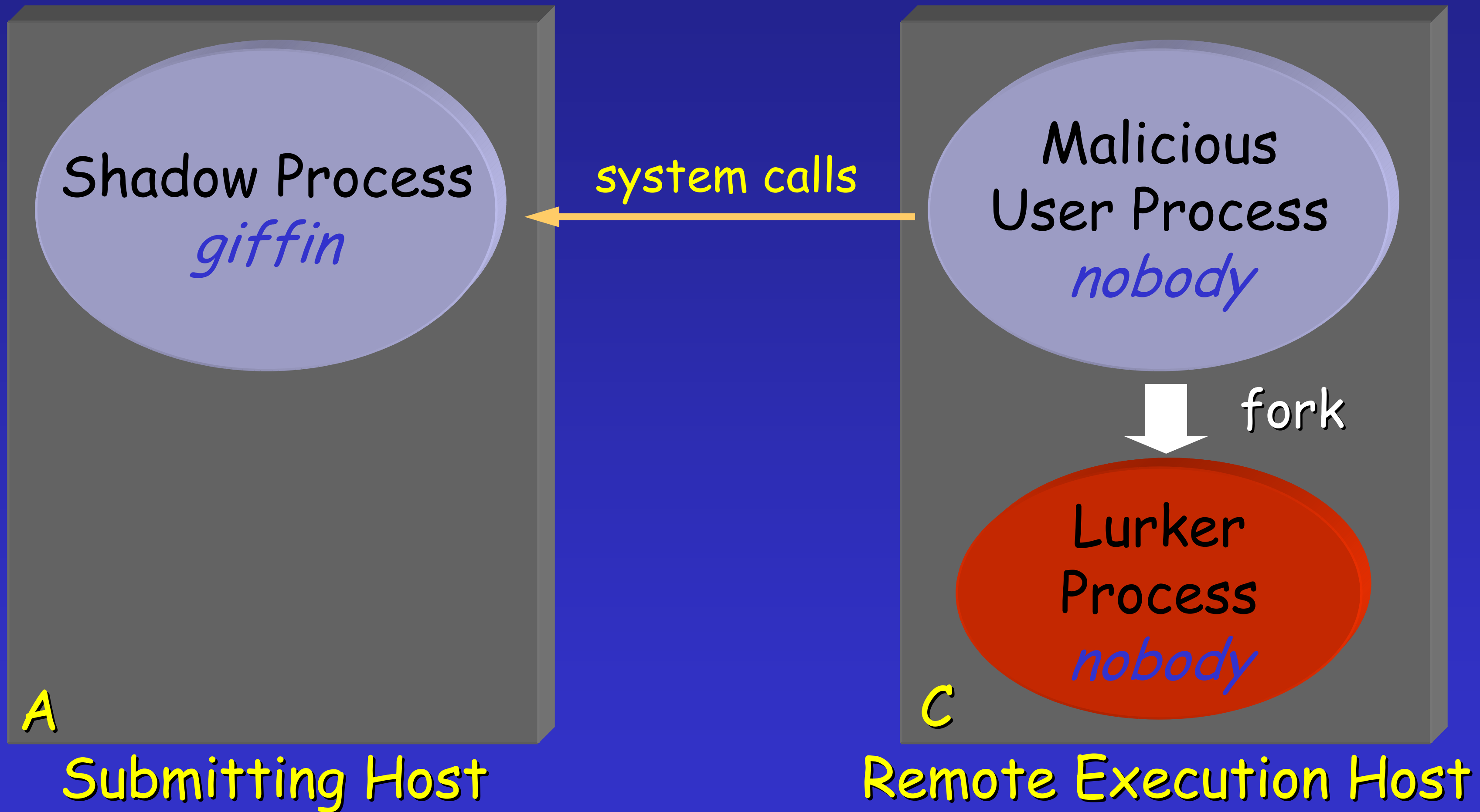


- Goal: Execute malicious code in the server

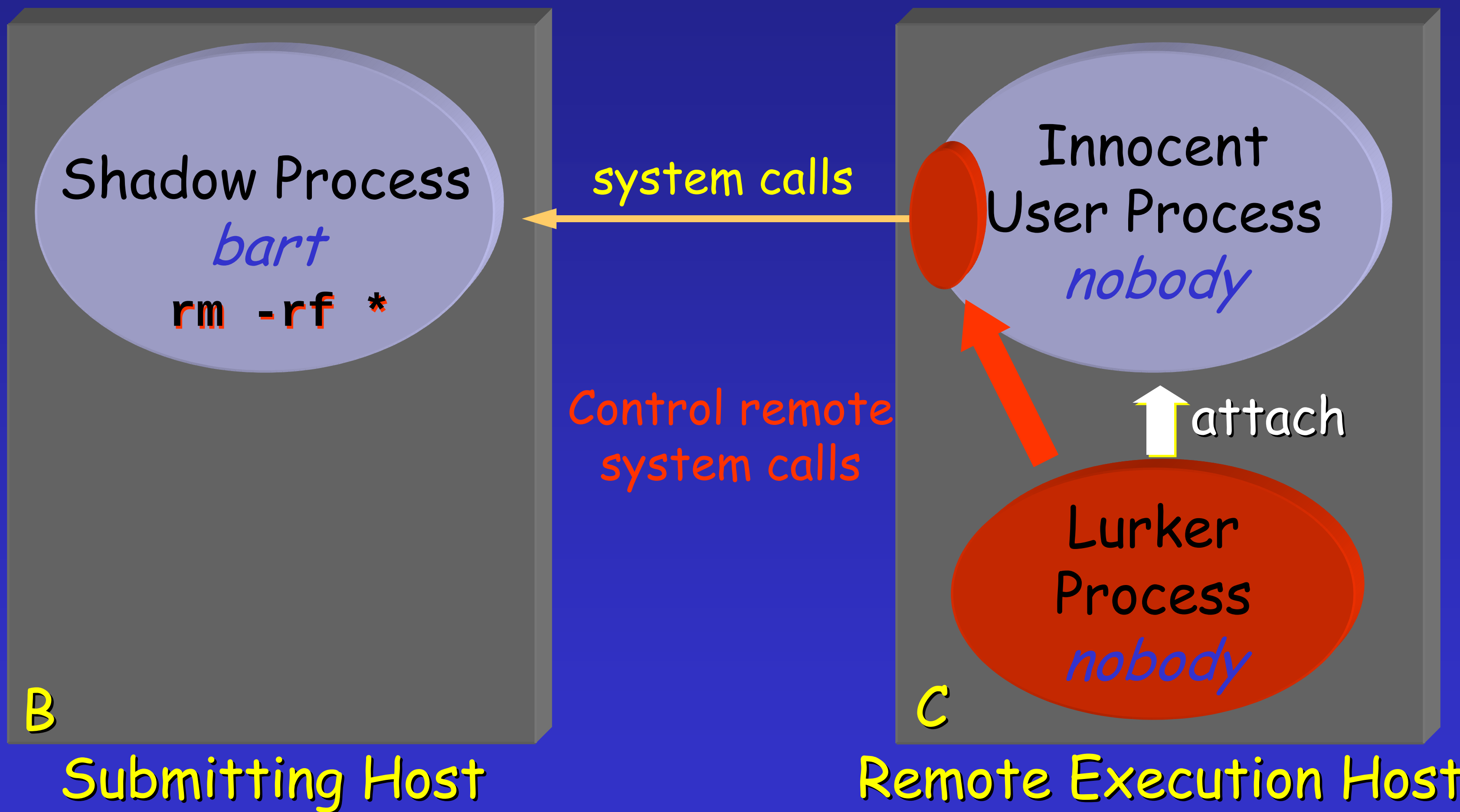
Example: Remote Execution Attack



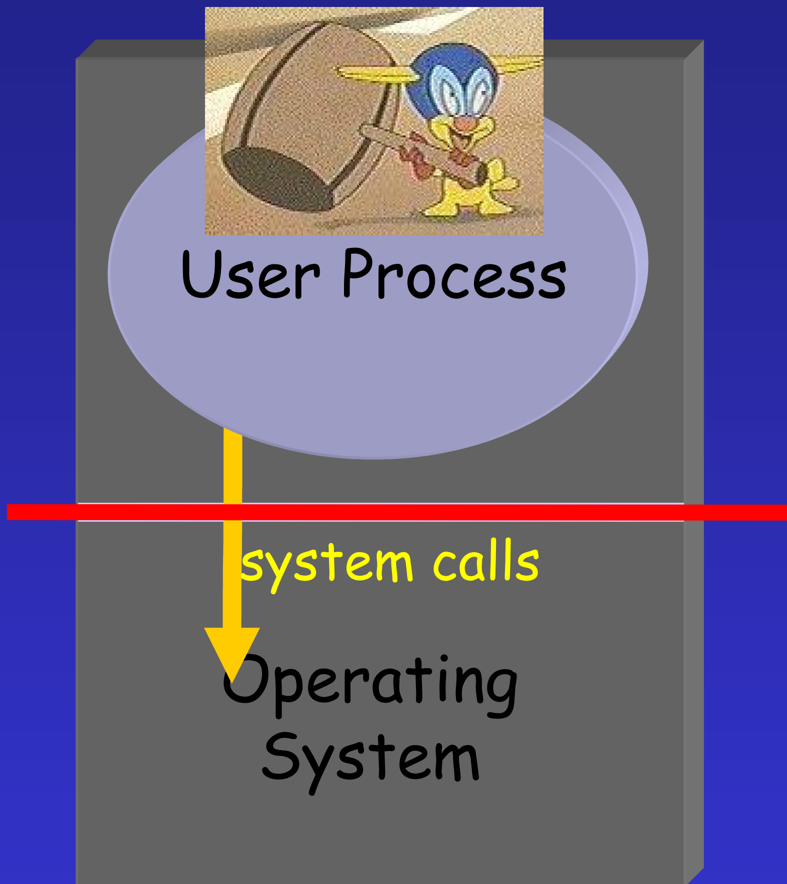
Example: Remote Execution Attack



Example: Remote Execution Attack

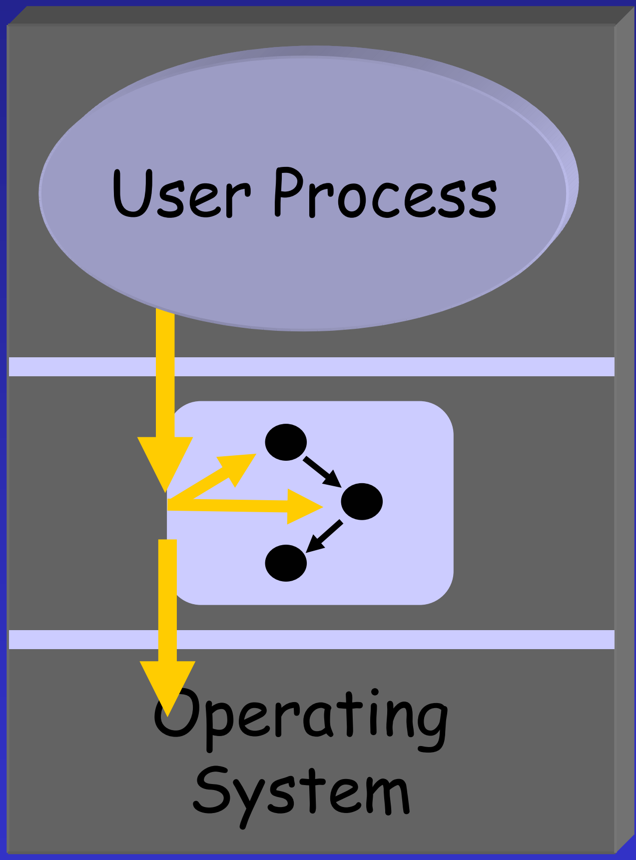


Our Objective



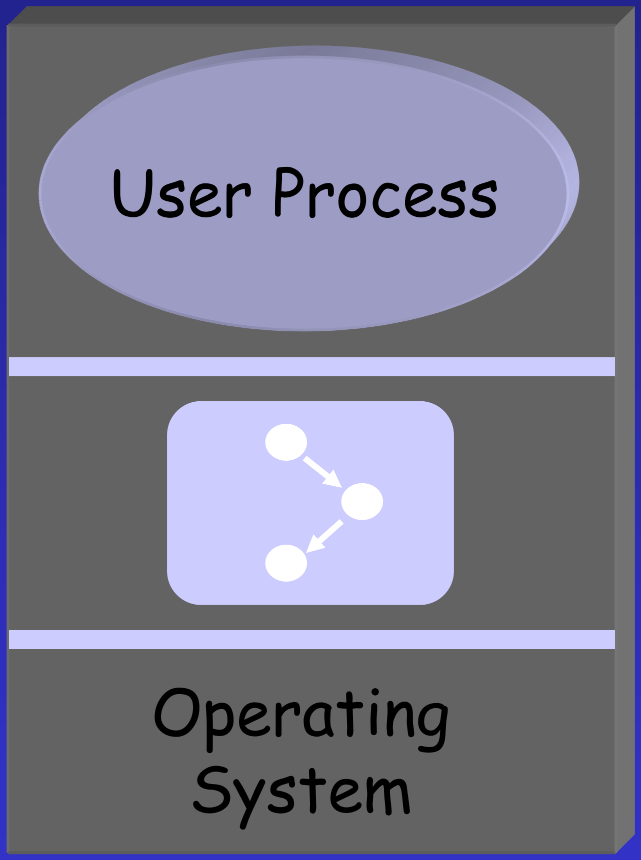
- Detect malicious activity before harm caused to local machine
- … before operating system executes malicious system call

Model-Based Intrusion Detection



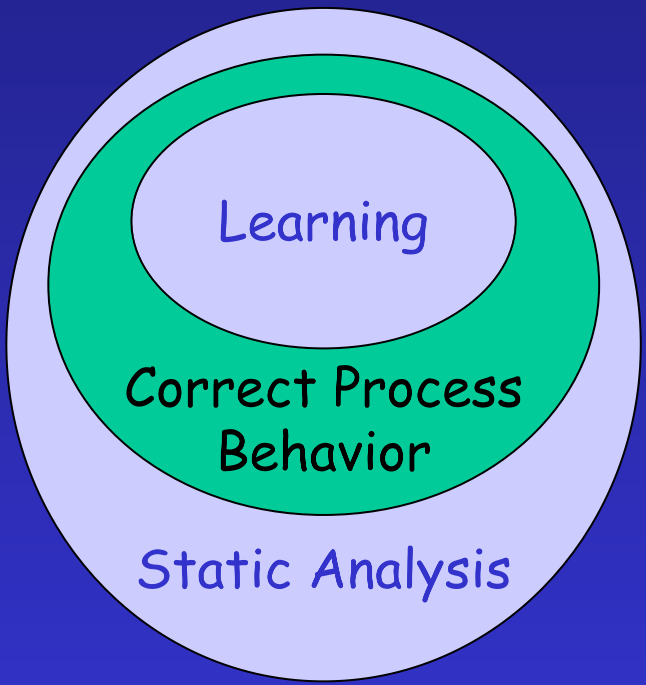
- Build model of correct program behavior
- Model: automaton specifying all valid system call sequences
- Runtime monitor ensures execution does not violate model

Model-Based Intrusion Detection



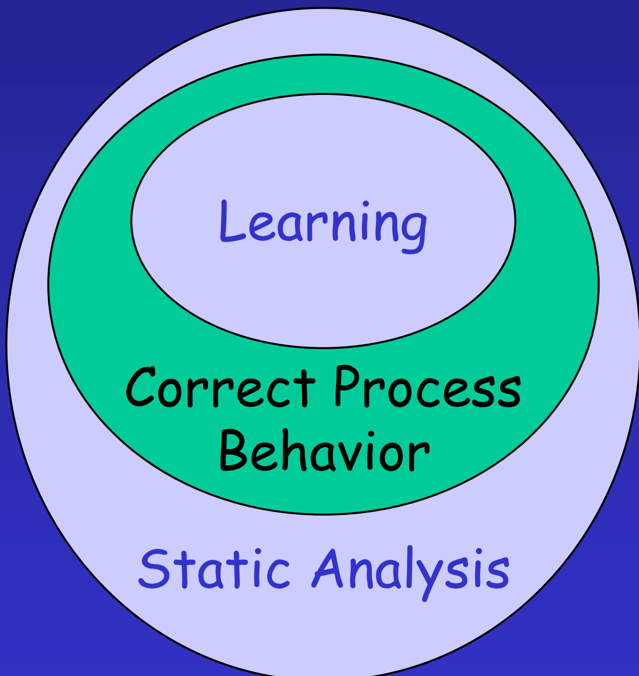
- Model must be fast to operate
- Model must accurately represent program
  - Context-sensitive models restrict impossible paths

Automated Model Construction



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

Automated Model Construction

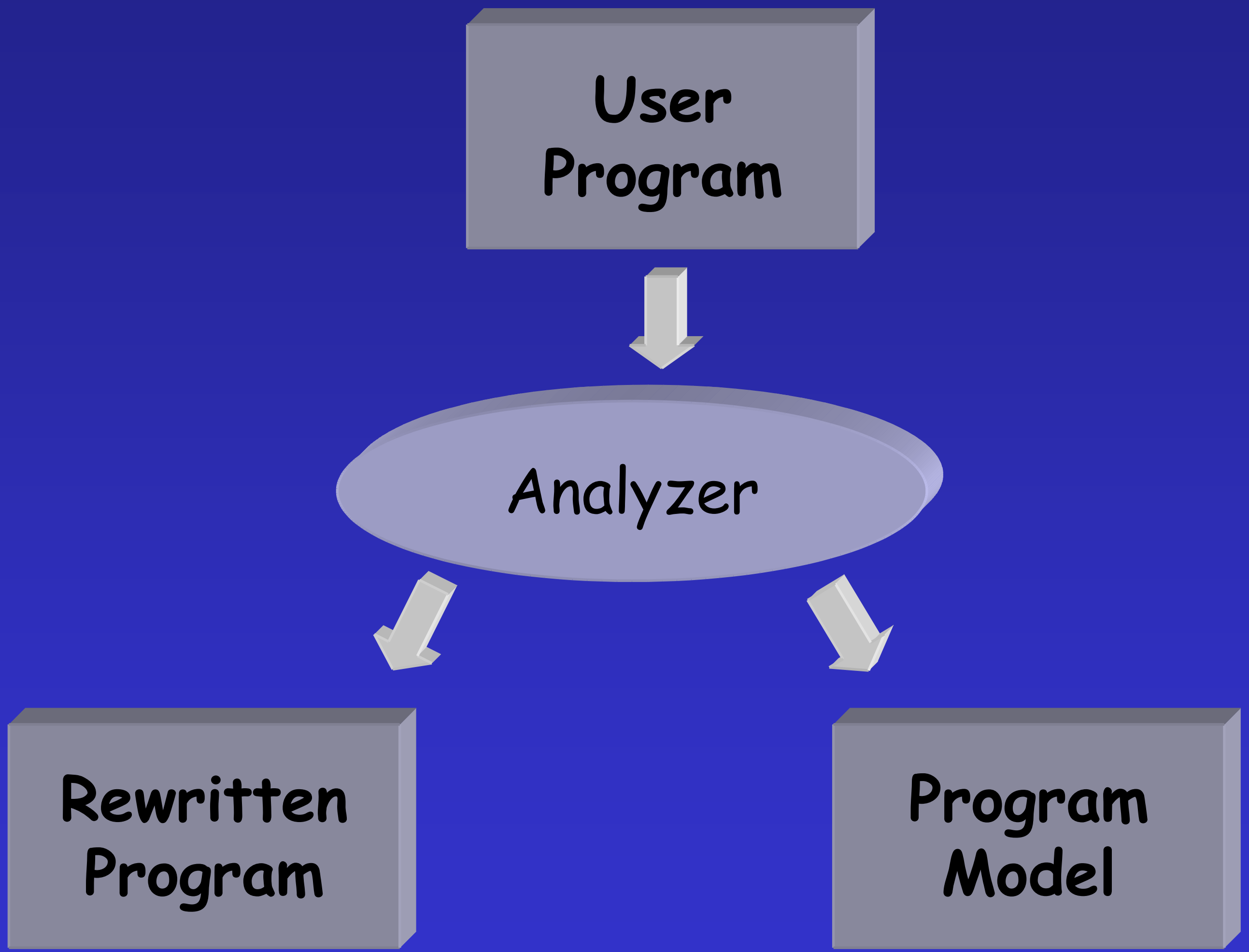


- Static analysis challenge
  - Design an efficient, accurate model
- Answers
  - Dyck model
  - Data flow analysis to recover arguments

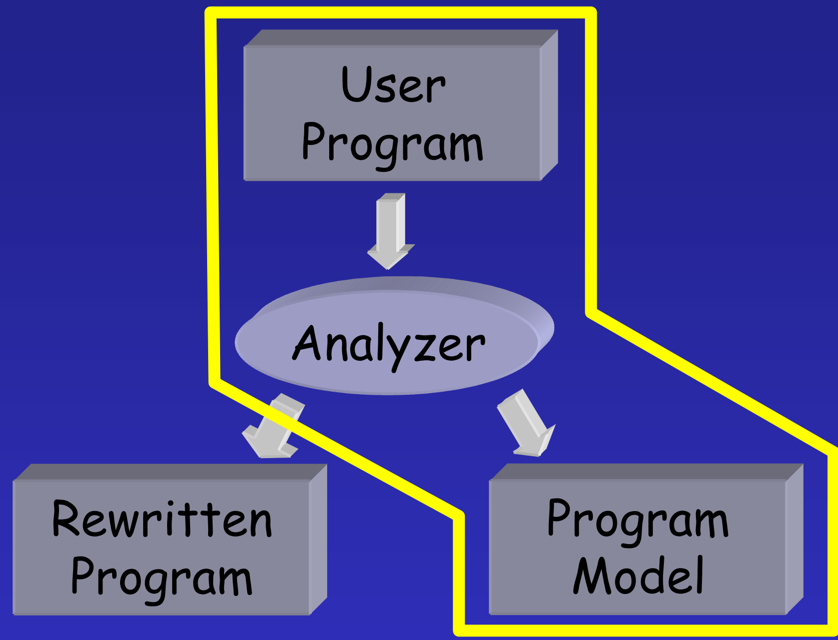
Our Approach

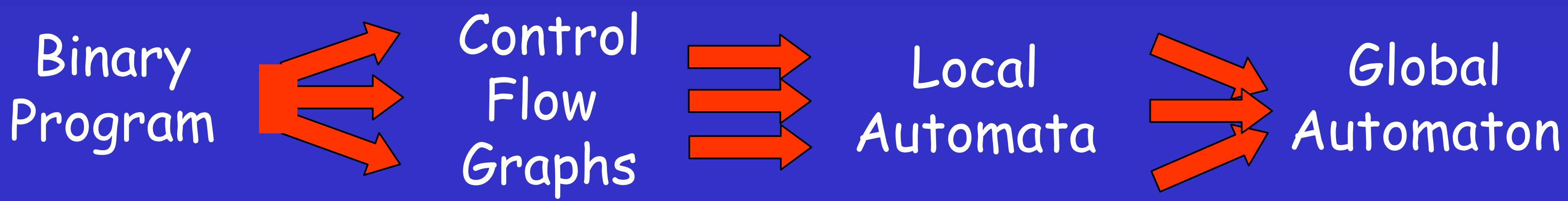
- Build model of correct program behavior
  - Static analysis of binary code
  - Construct an automaton modeling all system call sequences the program can generate
- Ensure execution does not violate model
  - Use automaton to monitor system calls.
  - If automaton reaches an invalid state, then an intrusion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

Model-Based Intrusion Detection



Model Construction

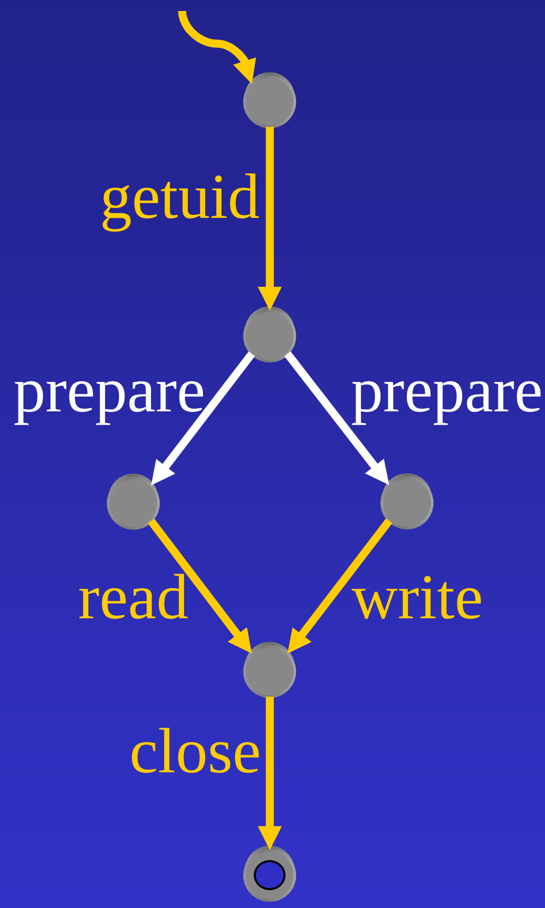




Code Example

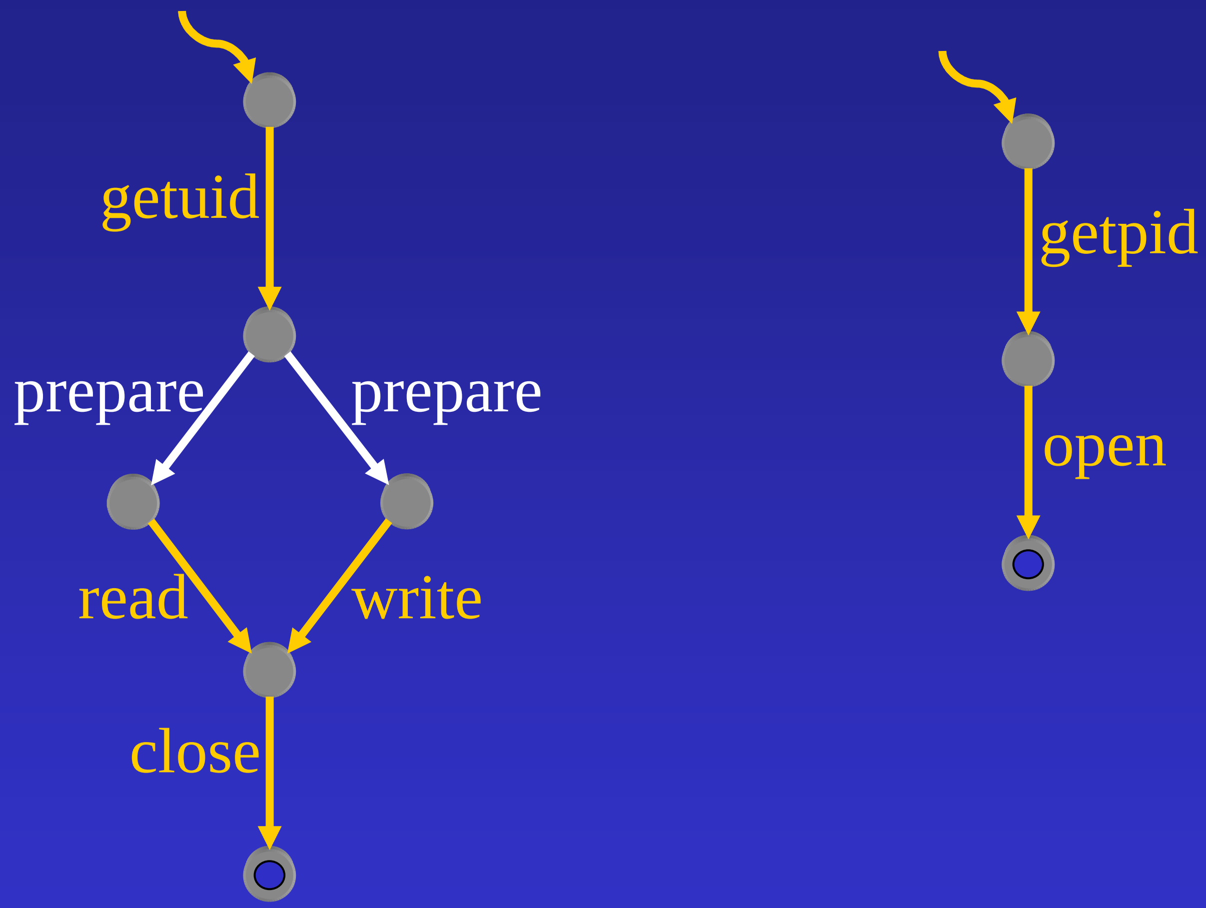
```
char *filename;
pid_t[2] pid;

int prepare (int index) {
  char buf[20];
  pid[index] = getpid();
  strcpy(buf, filename);
  return open(buf, O_RDWR);
}
```

Code Example

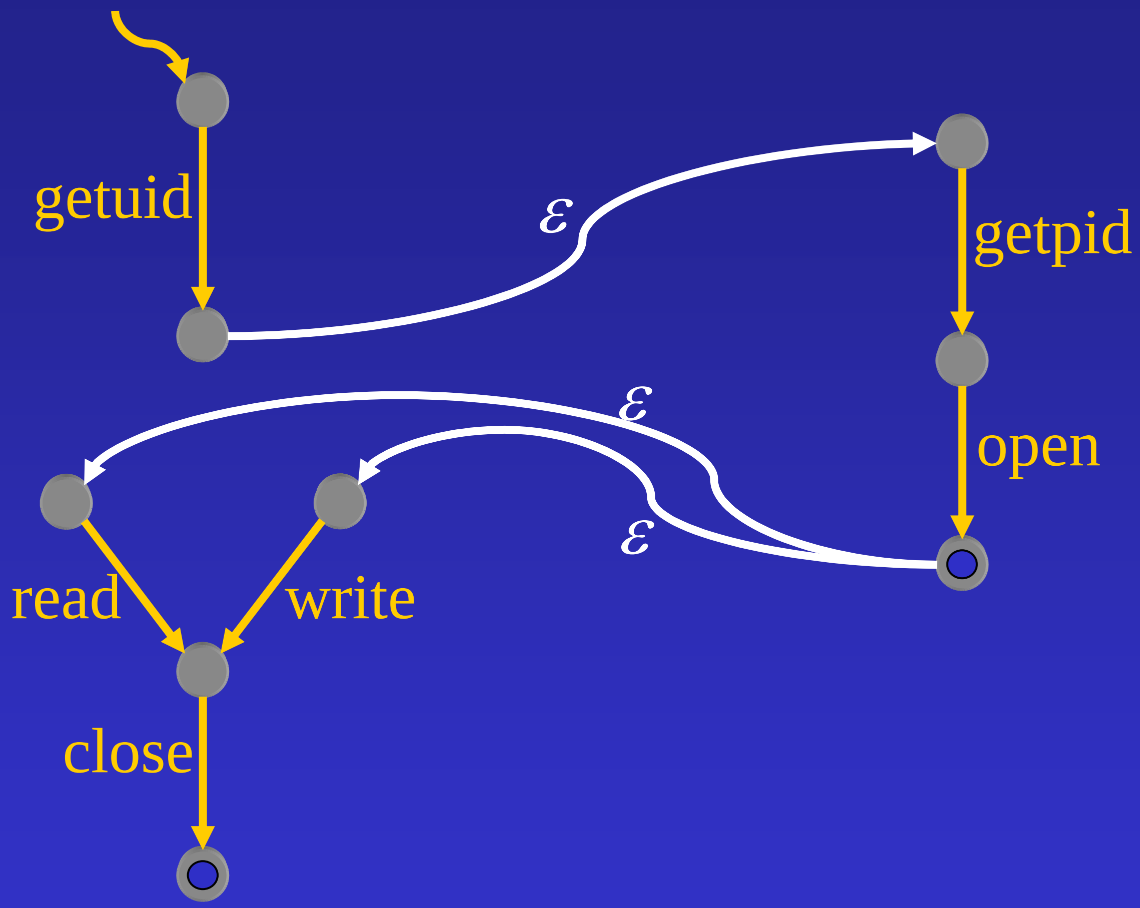



```
void action (void) {
  uid_t uid = getuid();
  int handle;

  if (uid != 0) {
    handle = prepare(1);
    read(handle, …);
  } else {
    handle = prepare(0);
    write(handle, …);
  }

  close(handle);
}
```

NFA Model

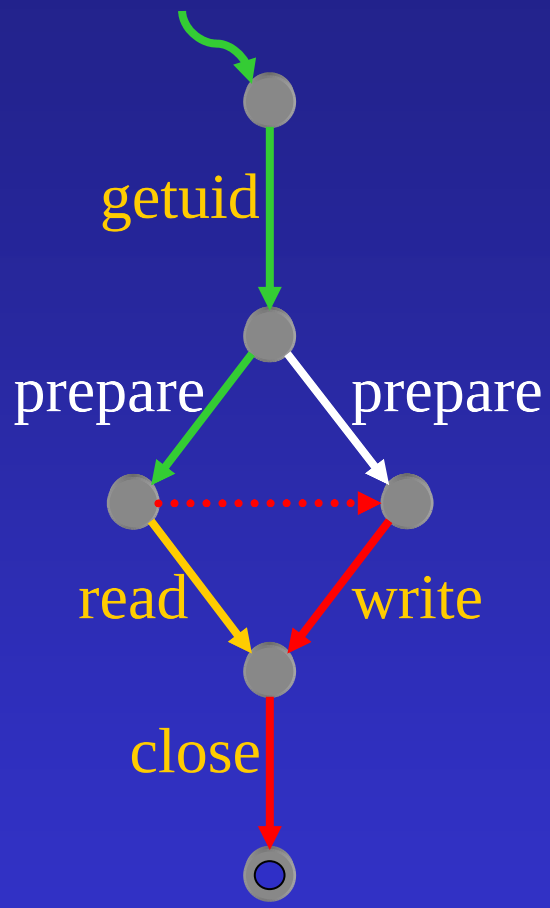


NFA Model



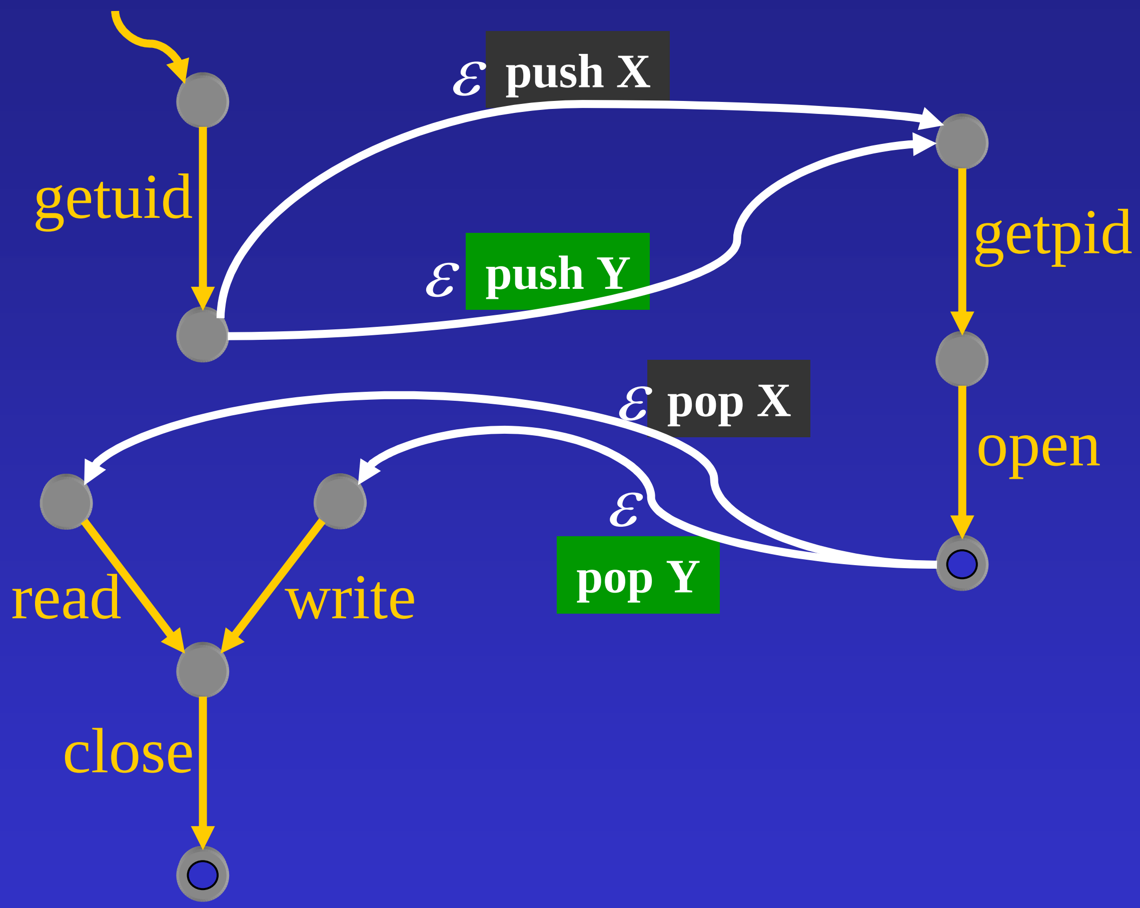
getuid
ε
getpid
open
ε
ε
read
write
close

Impossible Path Exploit

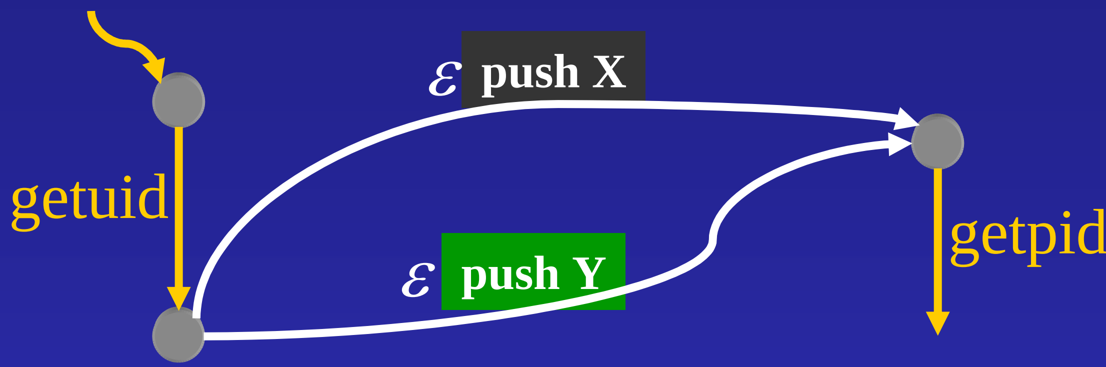



```
void action (void) {
    uid_t uid = getuid();
    int handle;

    if (uid != 0) {
        handle = prepare(1);
        read(handle, …);
    } else {
        handle = prepare(0);
        write(handle, …);
    }

    close(handle);
}
```

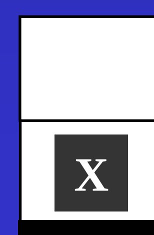
PDA Model


getuid
ε push X
ε push Y
getpid
open
ε pop X
ε
pop Y
read
write
close

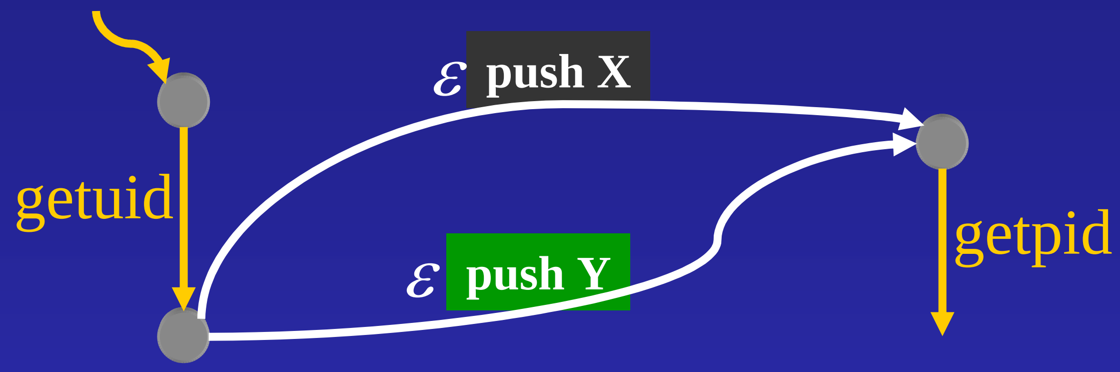
PDA Problems



- Impossible paths still exist
  - Non-determinism indicates missing execution information
- PDA run-time state explosion
  - ε-edge identifiers maintained on a stack
  - Stack non-determinism is expensive
  - post* algorithm: cubic in automaton size



PDA Problems



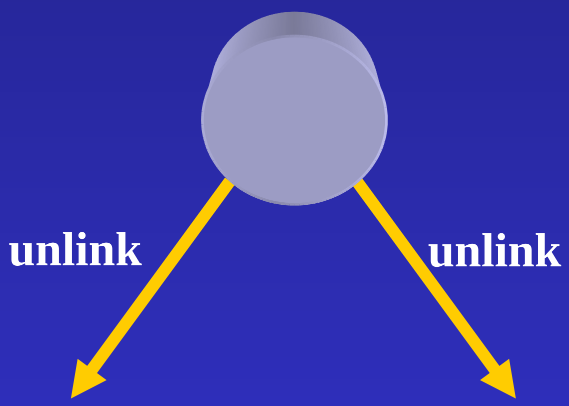
- Unusable as program model
  - Orders of magnitude slowing of application
    - [Wagner *et al.* 01, Giffin *et al.* 02]
  - Conclusion: only weaker NFA models have reasonable performance

Some Recent History…

		Model	Speed	Imp Paths
2001	Wagner	NFA	Moderate	Many
		PDA	Slow	Some
	Sekar	DFA	Fast	Many
2002	Giffin	NFA	Fast	Many
		BPDA	Moderate	Some
2003	Feng	VtPaths	Fast	Few
2004	Giffin	Dyck	Fast	Few
	Feng	VPStatic	Fast	Few

NFA

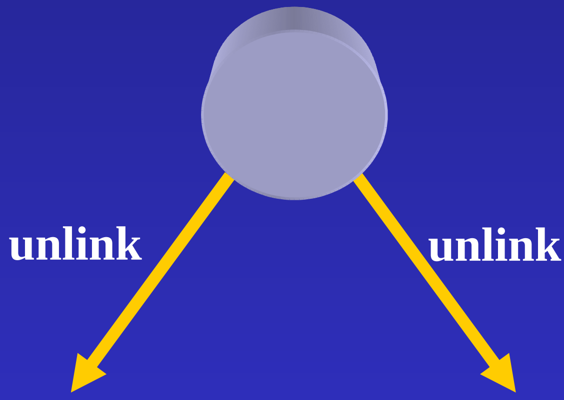
State non-determinism



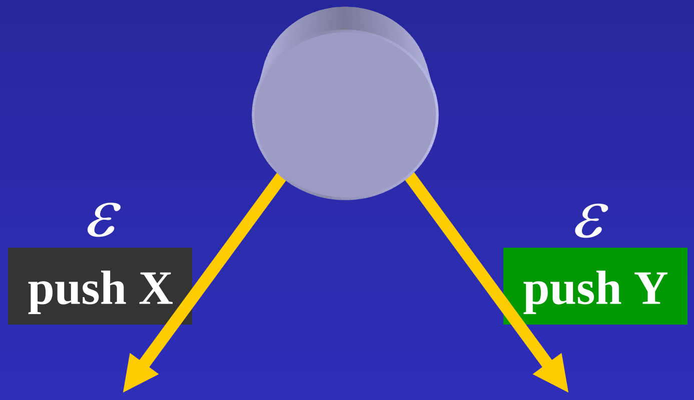
State non-determinism is cheap.

Non-Deterministic PDA

State non-determinism



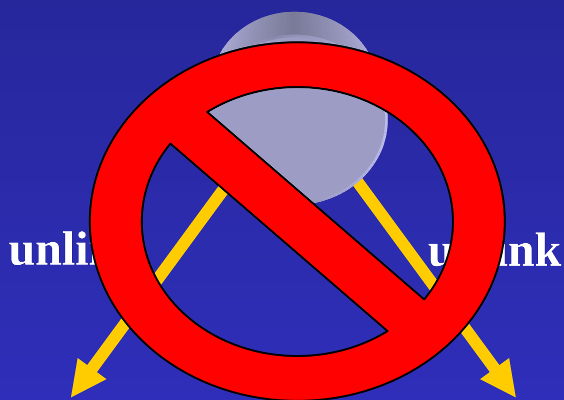
Stack non-determinism



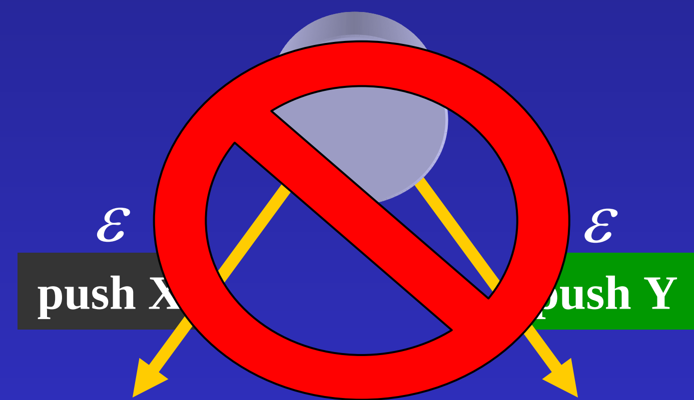
Stack non-determinism is expensive.

Deterministic PDA (DPDA)

State non-determinism

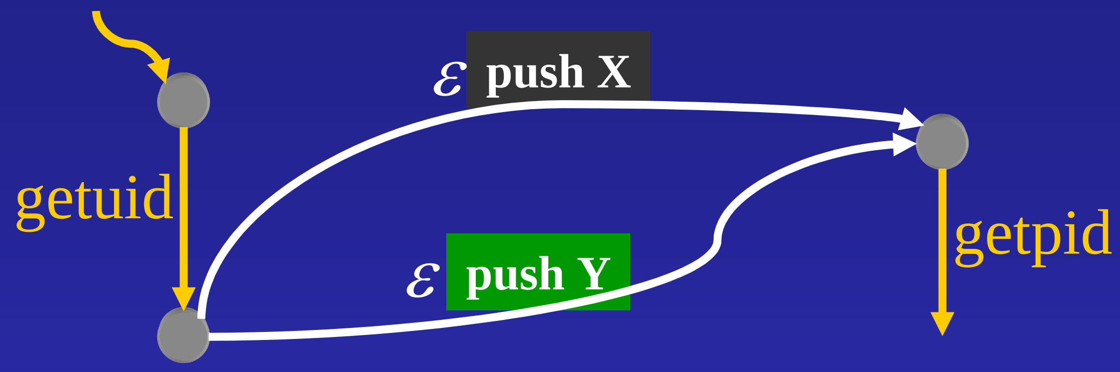


Stack non-determinism



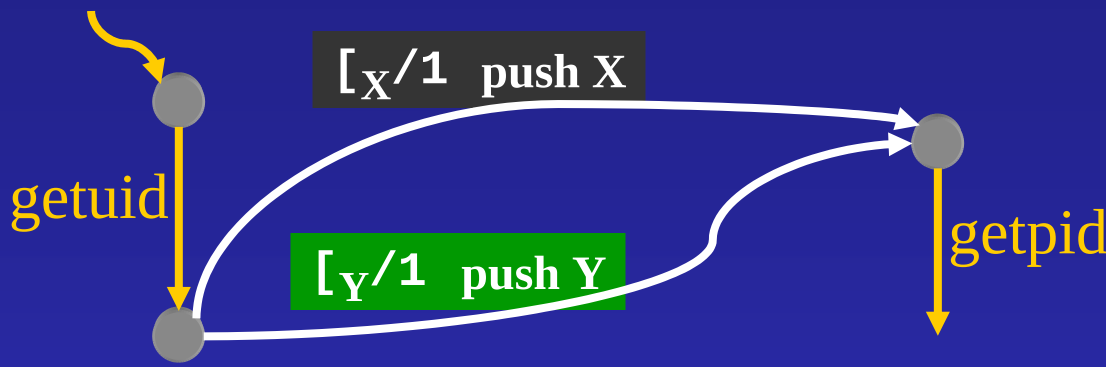
- Model exposes stack operations & target states
- Possible exponential increase in model size

Deterministic PDA (DPDA)



- Replace ε-edges with symbol describing stack operation & target state

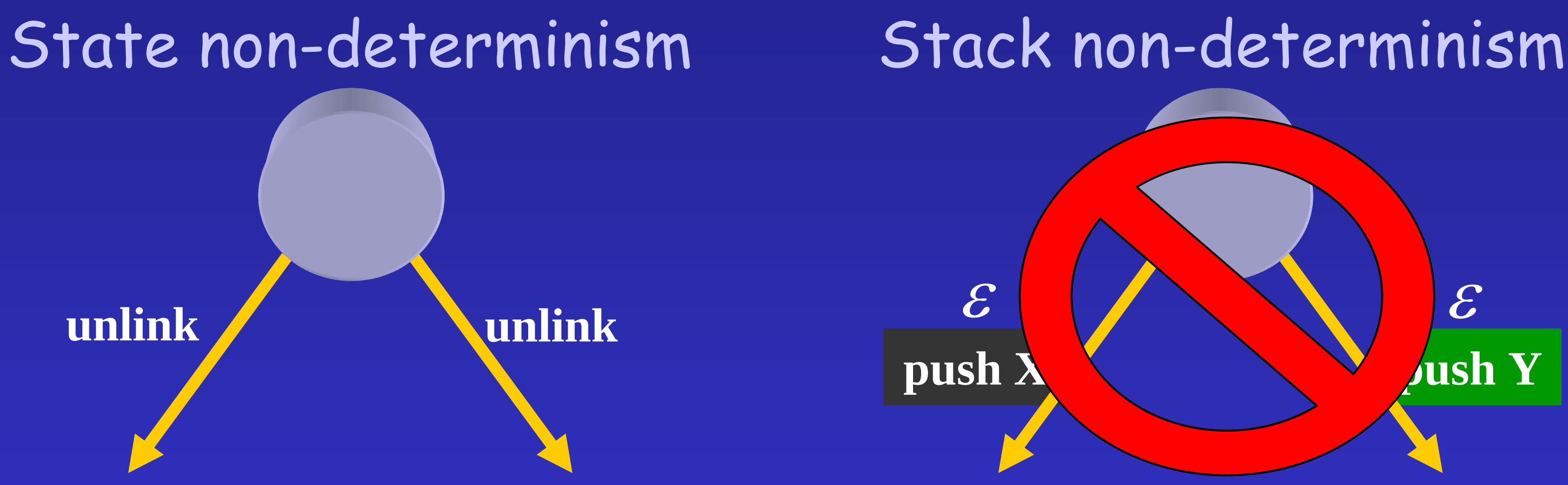
Deterministic PDA (DPDA)



- Replace ε-edges with symbol describing stack operation & target state
- Input symbol describes:
  - $[_X$: How to update stack
  - /1: How to traverse automaton transitions

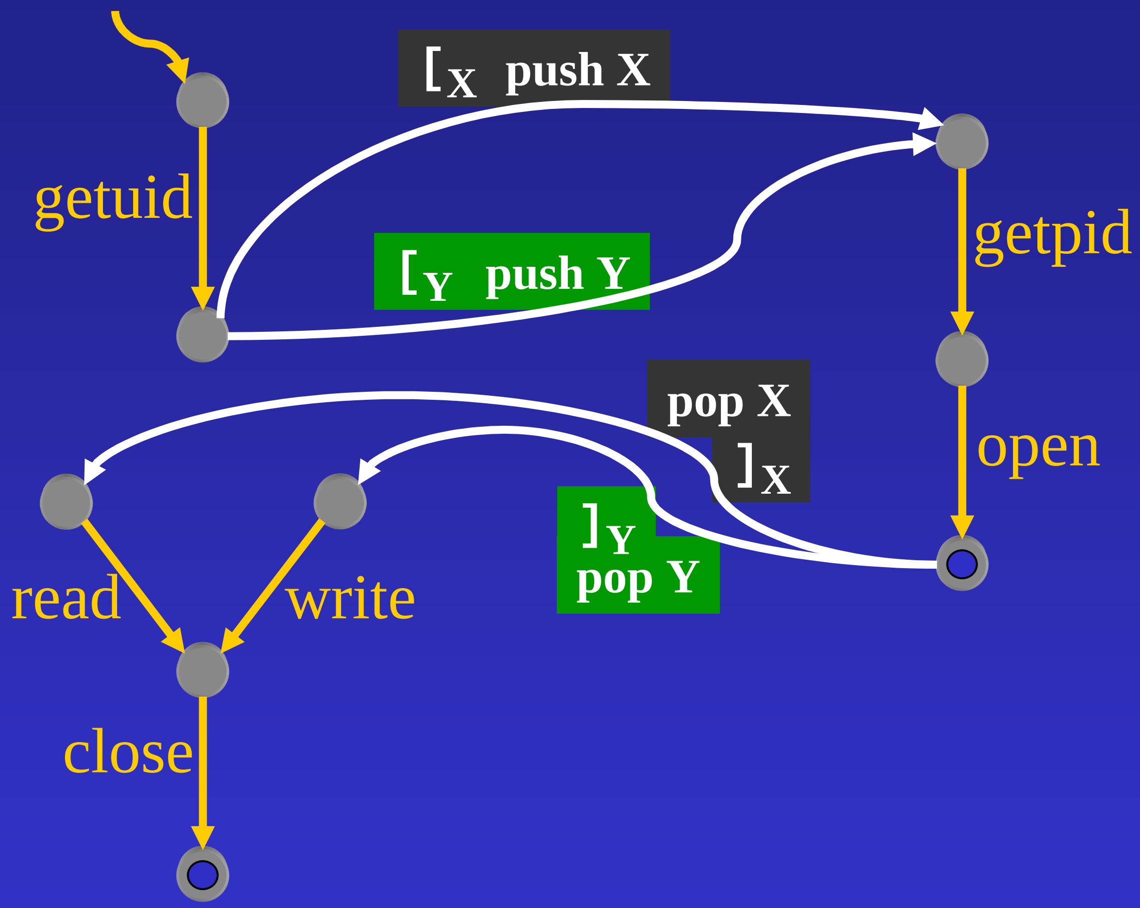
Stack-Deterministic PDA (sDPDA)

Dyck Model



- Model exposes stack operations
- No increase in model size

Stack-Deterministic PDA (sDPDA)

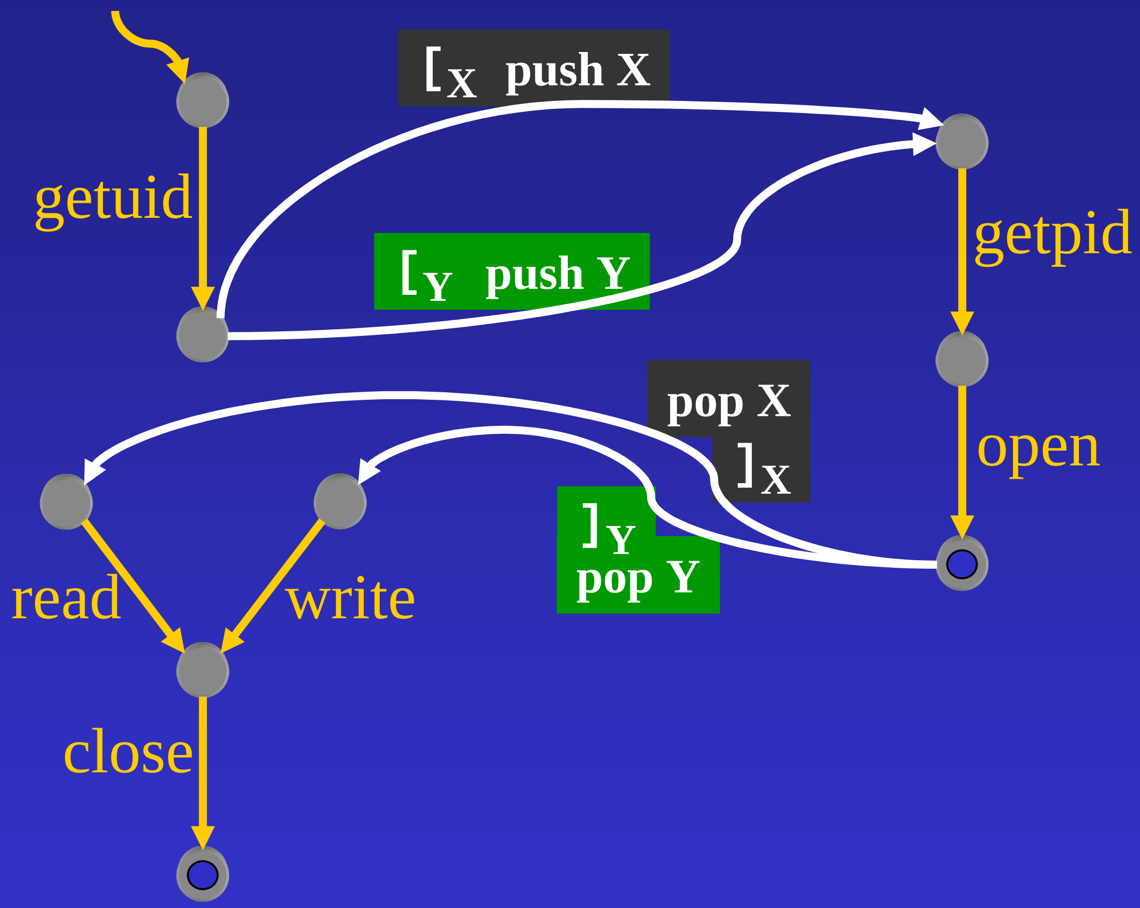


Input Symbol Processing Complexity

Model	*Time Complexity*	*Space Complexity*	*Input Alphabet Size*
PDA	$O(nm^2)$	$O(nm^2)$	k
DPDA	$O(1)$	$O(1)$	$\Theta(knr)$
sDPDA	$O(n)$	$O(n)$	$\Theta(kr)$

- n is state count
- m is transition count
- k is PDA input alphabet size
- r is PDA stack alphabet size

Dyck Model



$[_X$ push X
getuid
$[_Y$ push Y
getpid
pop X
$]_X$
open
$]_Y$
pop Y
read
write
close

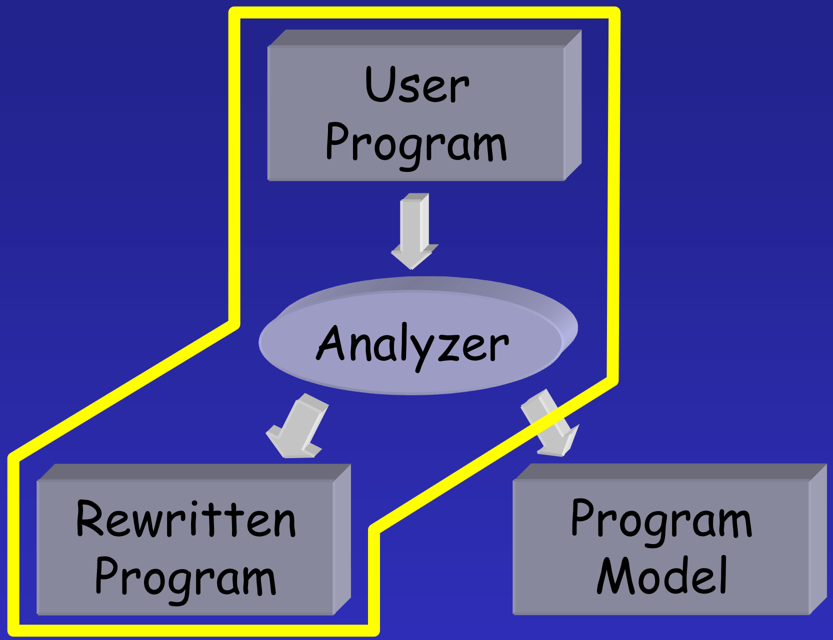
Dyck Model

getuid $[_X$ getpid open $]_X$ read close

getuid $[_y$ getpid open $]_y$ write close

- Matching brackets are alphabet symbols
  - Expose stack operations to runtime monitor
  - Language of bracket symbols is a Dyck language
  - Rewrite binary to generate bracket symbols

Binary Rewriting



Binary Program → Rewritten Binary

Determinizing via Binary Rewriting

- Insert code to generate bracket symbols around function call sites
- Notify monitor of stack activity
- Determinizes stack operations

```
void action (void) {
  uid_t uid = getuid();
  int handle;

  if (uid != 0) {
    precall(X);
    handle = prepare(1);
    postcall(X);
    read(handle, …);
  } else {
    precall(Y);
    handle = prepare(0);
    postcall(Y);
    write(handle, …);
  }

  close(handle);
}
```

Determinizing via Binary Rewriting

- Dyck null calls meaningful only when **prepare** generates system calls

Relevant:

$\ldots\ [_X\ \texttt{write}\ ]_X\ \ldots$

Irrelevant:

$\ldots\ [_X\ ]_X\ \ldots$

```
void action (void) {
  uid_t uid = getuid();
  int handle;

  if (uid != 0) {
    precall(X);
    handle = prepare(1);
    postcall(X);
    read(handle, …);
  } else {
    precall(Y);
    handle = prepare(0);
    postcall(Y);
    write(handle, …);
  }

  close(handle);
}
```

Determinizing via Binary Rewriting

- Maintain history stack in rewritten binary
- Records null calls encountered since last system call

Relevant:

… $[_X$ write $]_X$ …

Irrelevant:

… $[_X$ $]_X$ …

```
void action (void) {
  uid_t uid = getuid();
  int handle;

  if (uid != 0) {
    precall(X);
    handle = prepare(1);
    postcall(X);
    read(handle, …);
  } else {
    precall(Y);
    handle = prepare(0);
    postcall(Y);
    write(handle, …);
  }

  close(handle);
}
```

Null Call Squelching

Relevant:

… $[_X$ write $]_X$ …

History Stack:

$]_X$ $]_B$ $[_C$ $[_X$

Null Call Squelching

Irrelevant:

… $[_X$ $]_X$ …

History Stack:

$]_A$ $]_B$ $[_C$ $[_X$ $]_X$

Squelching bounds number of null calls produced

Determinizing via Stackwalks

- Recover stored return values by walking the call stack of the running process

Current: **`0x1003c, 0x318f0, 0x22cd8`**

Determinizing via Stackwalks

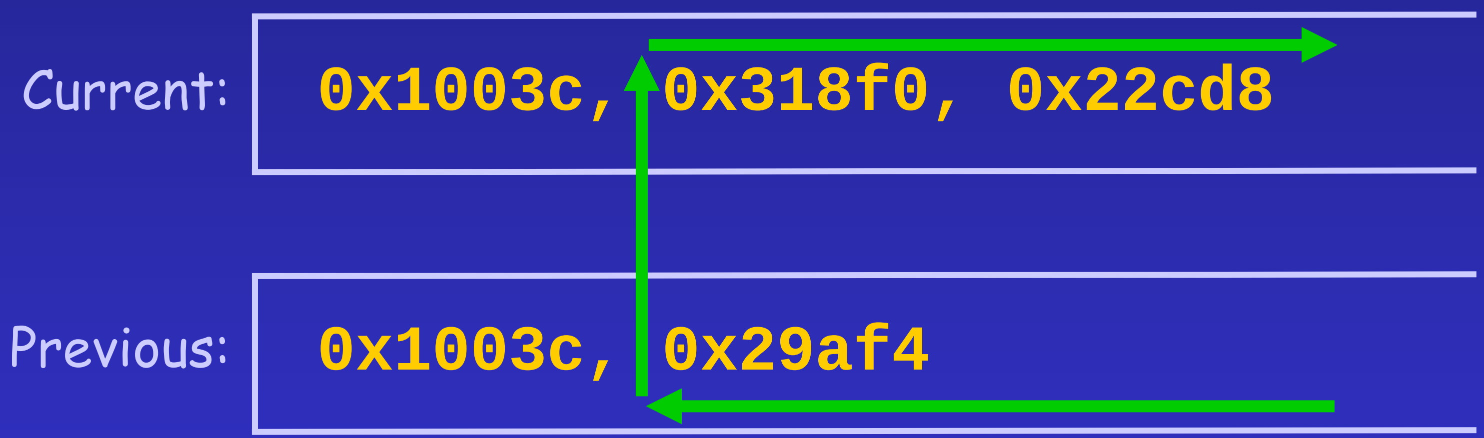
- Recover stored return values by walking the call stack of the running process

Current:	**0x1003c, 0x318f0, 0x22cd8**
Previous:	**0x1003c, 0x29af4**

- Compare to previous stack to generate pop and push input symbols
- Requires no binary rewriting

Determinizing via Stackwalks

- Recover stored return values by walking the call stack of the running process

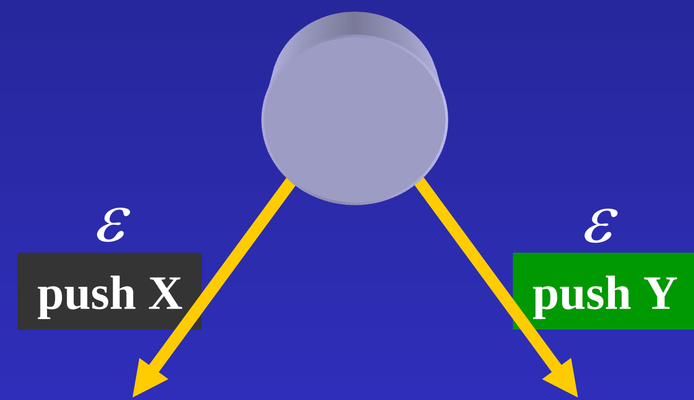


- Compare to previous stack to generate pop and push input symbols
- Requires no binary rewriting

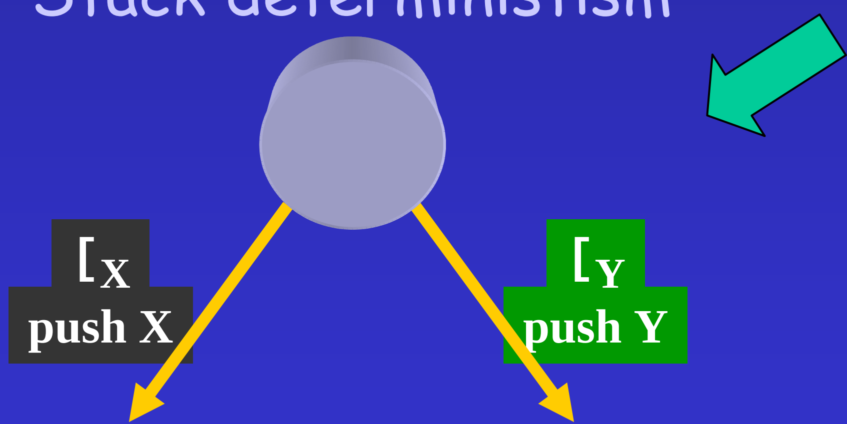
Dyck Model

Dyck model stack-determinizes PDA

Stack non-determinism



Stack deterministism



Only one valid stack configuration

Study 1: Dyck Model

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

Runtime Overheads

Execution times in se

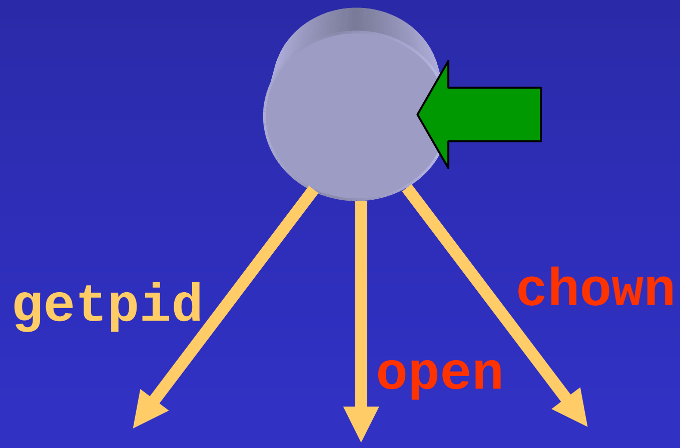
- Squelching removed 7.3 million symbols

Program	Base	NFA	Increase	Dyck	I…ease
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%

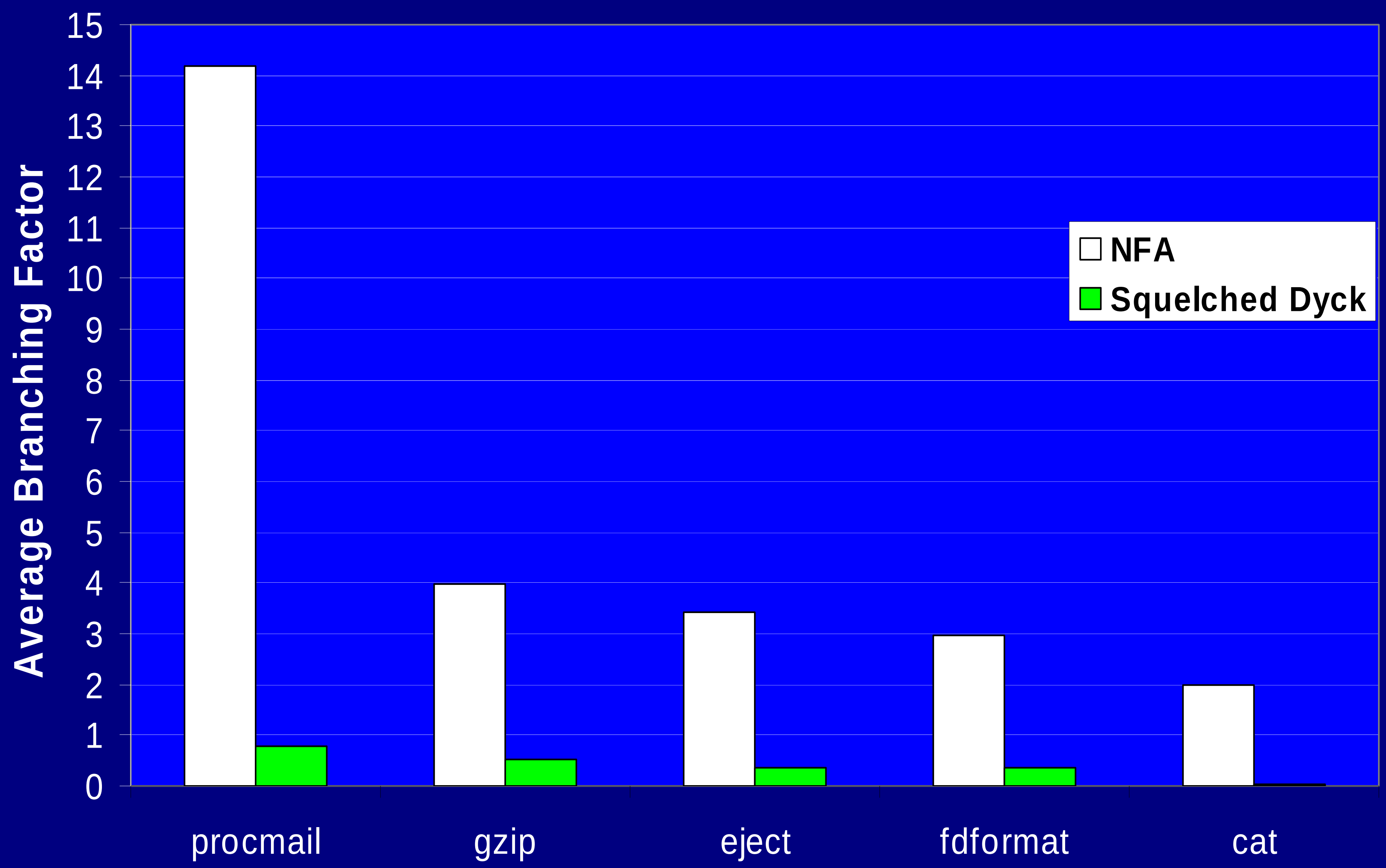
- High null call count
- Workload specific

Accuracy Metric

- Average branching factor

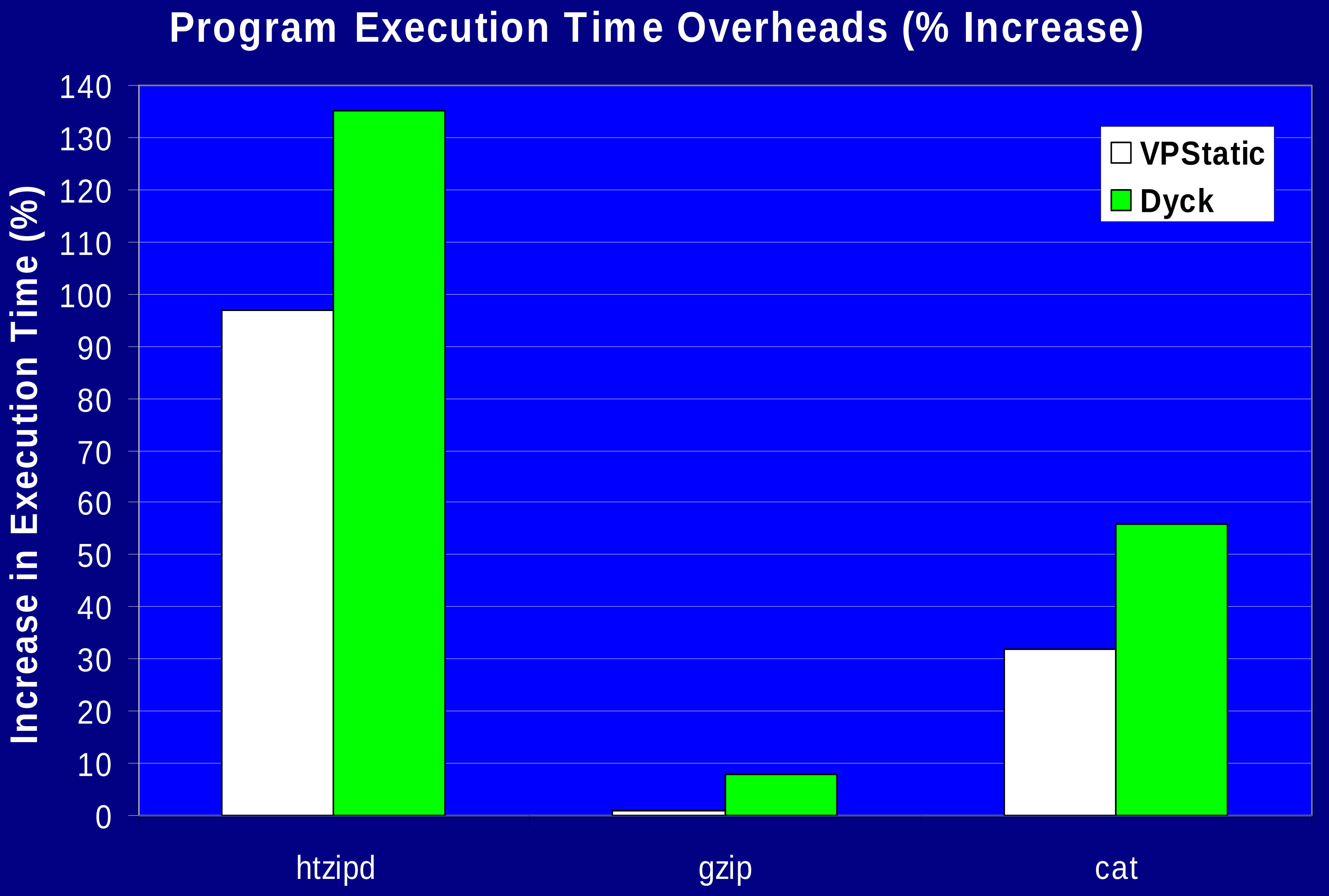


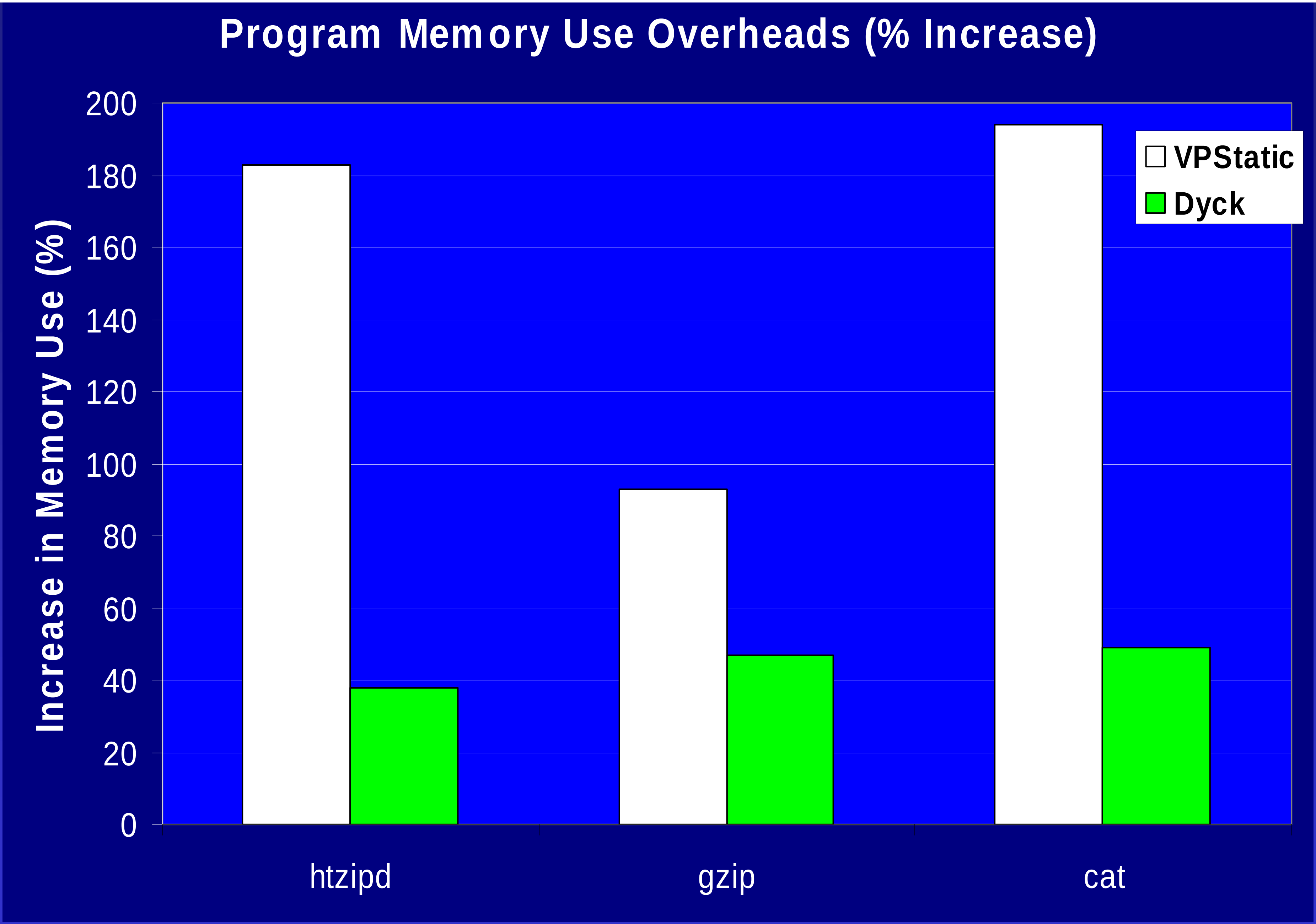
NFA and Dyck Model Accuracy



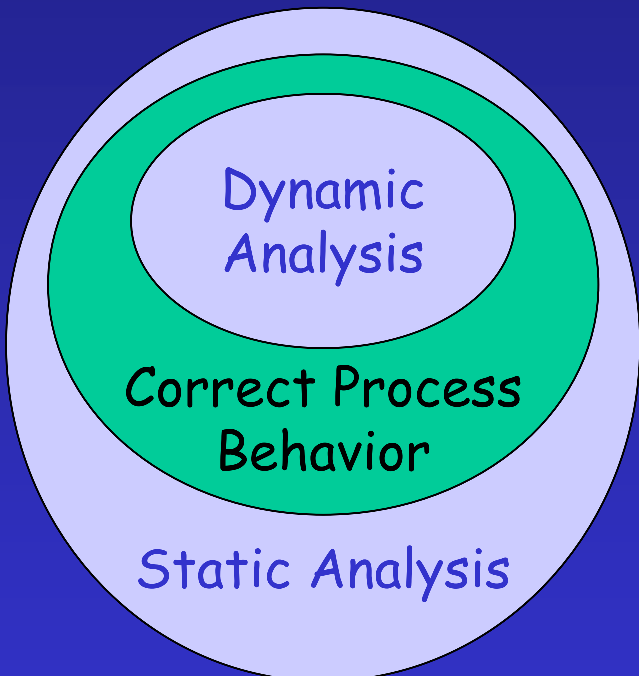
Study 2: DPDA vs. sDPDA

Program	Number of Instructions
htzipd	110,096
gzip	57,271
cat	52,601


Program Execution Time Overheads (% Increase)
Increase in Execution Time (%)
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
VPStatic
Dyck
htzipd
gzip
cat

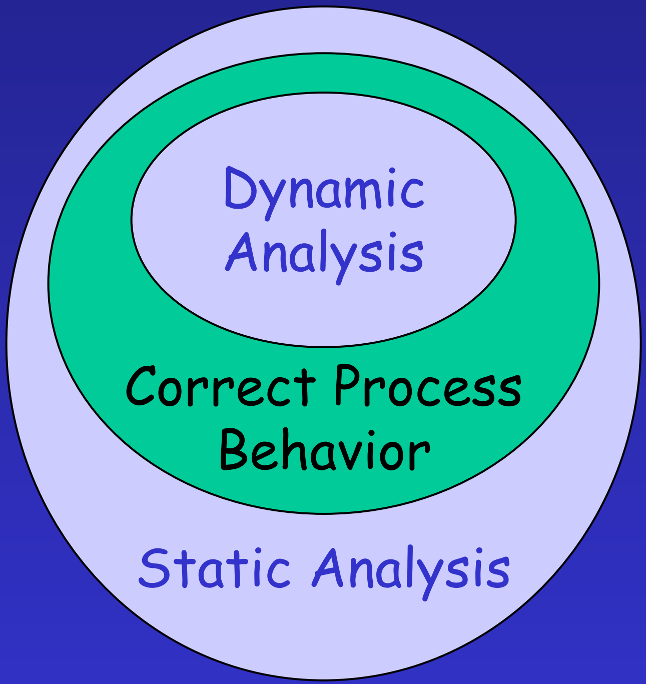

Program Memory Use Overheads (% Increase)
Increase in Memory Use (%)
200
180
160
140
120
100
80
60
40
20
0
VPStatic
Dyck
htzipd
gzip
cat

Analysis Combinations



- Static analysis
  - Conservative, nearly sound
  - Incorporating configuration information requires expensive analyses
- Dynamic analysis
  - Under-approximation produces false alarms
  - Reveals how configuration settings affect execution

Analysis Combinations



- Combined model
  - Dynamic: identify system call arguments
  - Dynamic: identify program branch behavior
  - Static: build Dyck model with added restrictions from dynamic analyses
- Joint work with Wenke Lee

Important Ideas

- Formalizing program models facilitates understanding & comparison.

- Exposing additional program state improves monitoring speed & model accuracy.