Stripped Binaries and Compiler Identification

Nathan Rosenblum

Paradyn Project

Paradyn / Dyninst Week

College Park, Maryland

April 27 - 28, 2009

But first…

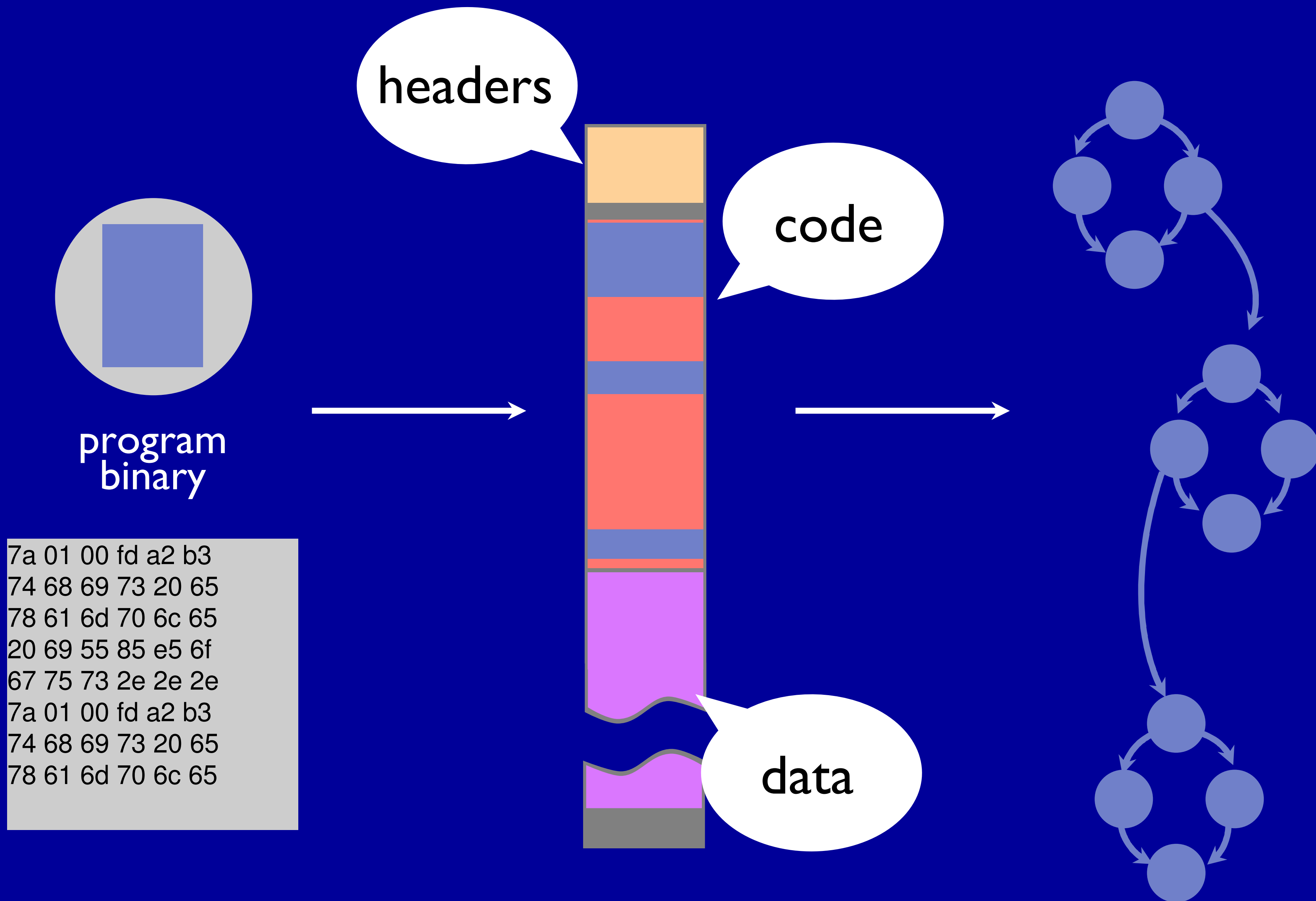
“ Binary program provenance ”

We started with:

Stripped binary parsing → Machine learning approaches → Compiler-specific problems arise → Compiler identification → …?

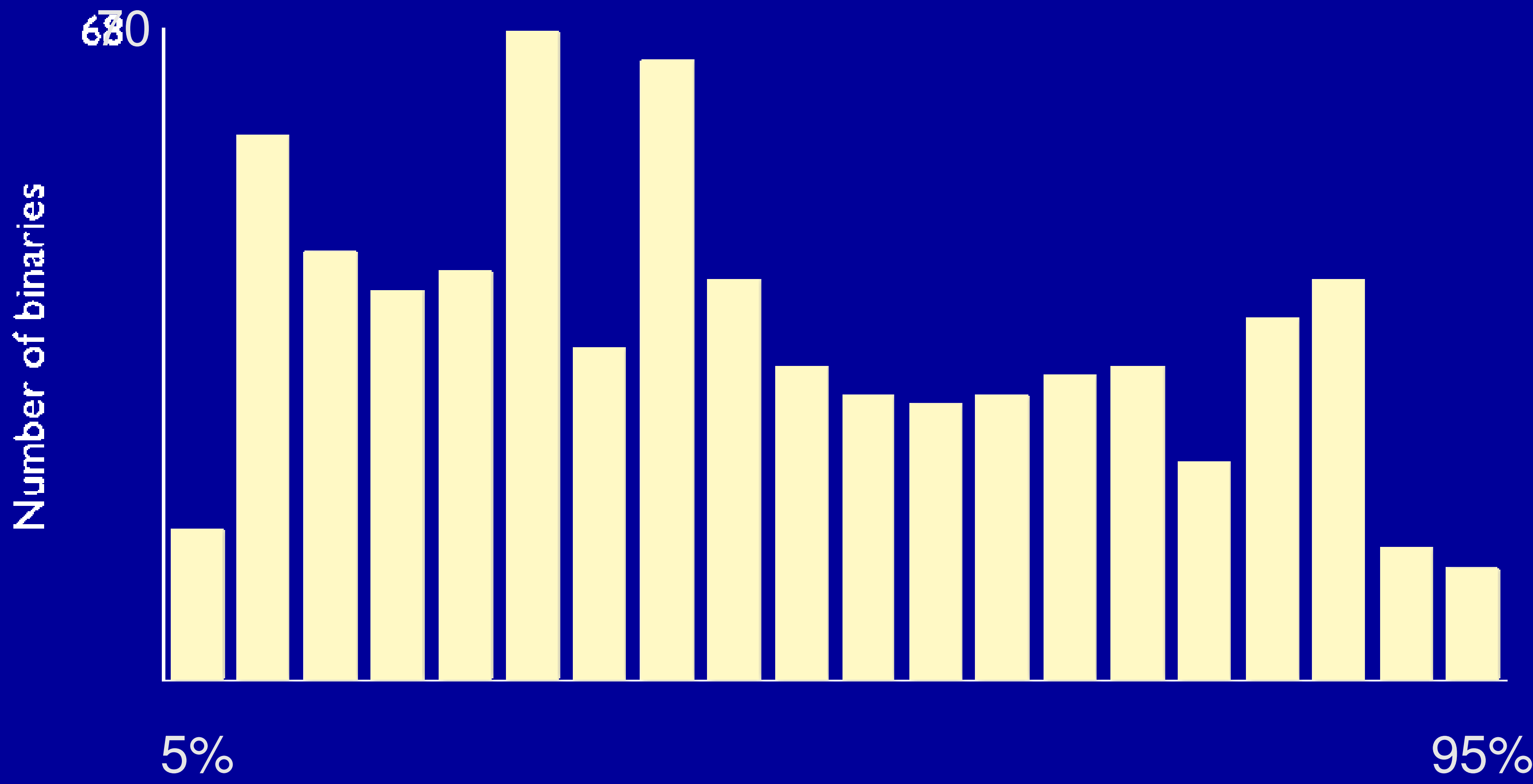
(Paradyn week 2008)

Parsed binaries are convenient...



Complete parsing is hard

Fraction of functions in “gap regions” after static parsing



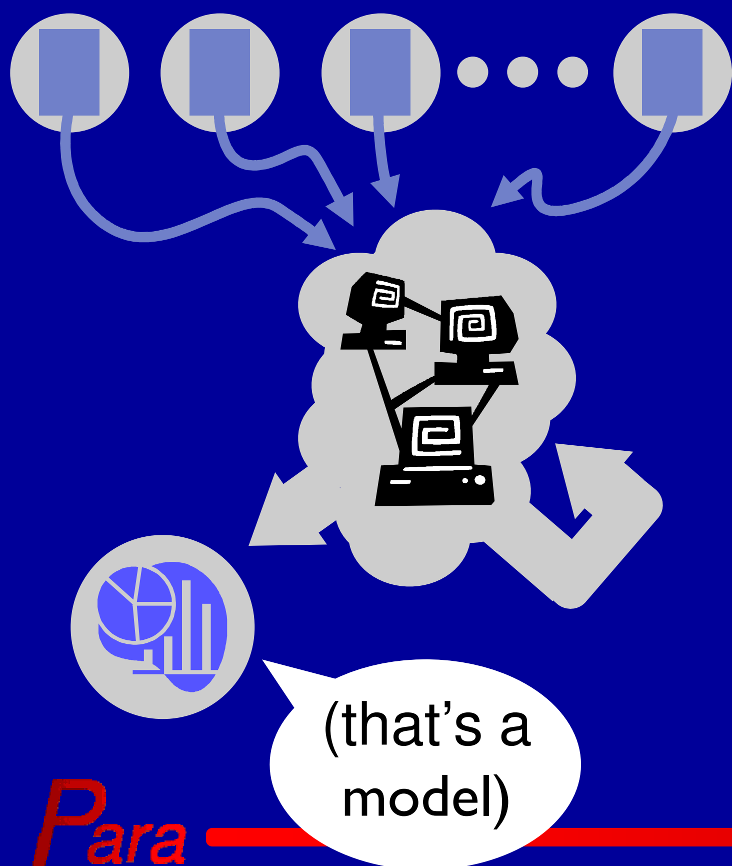
Getting full coverage

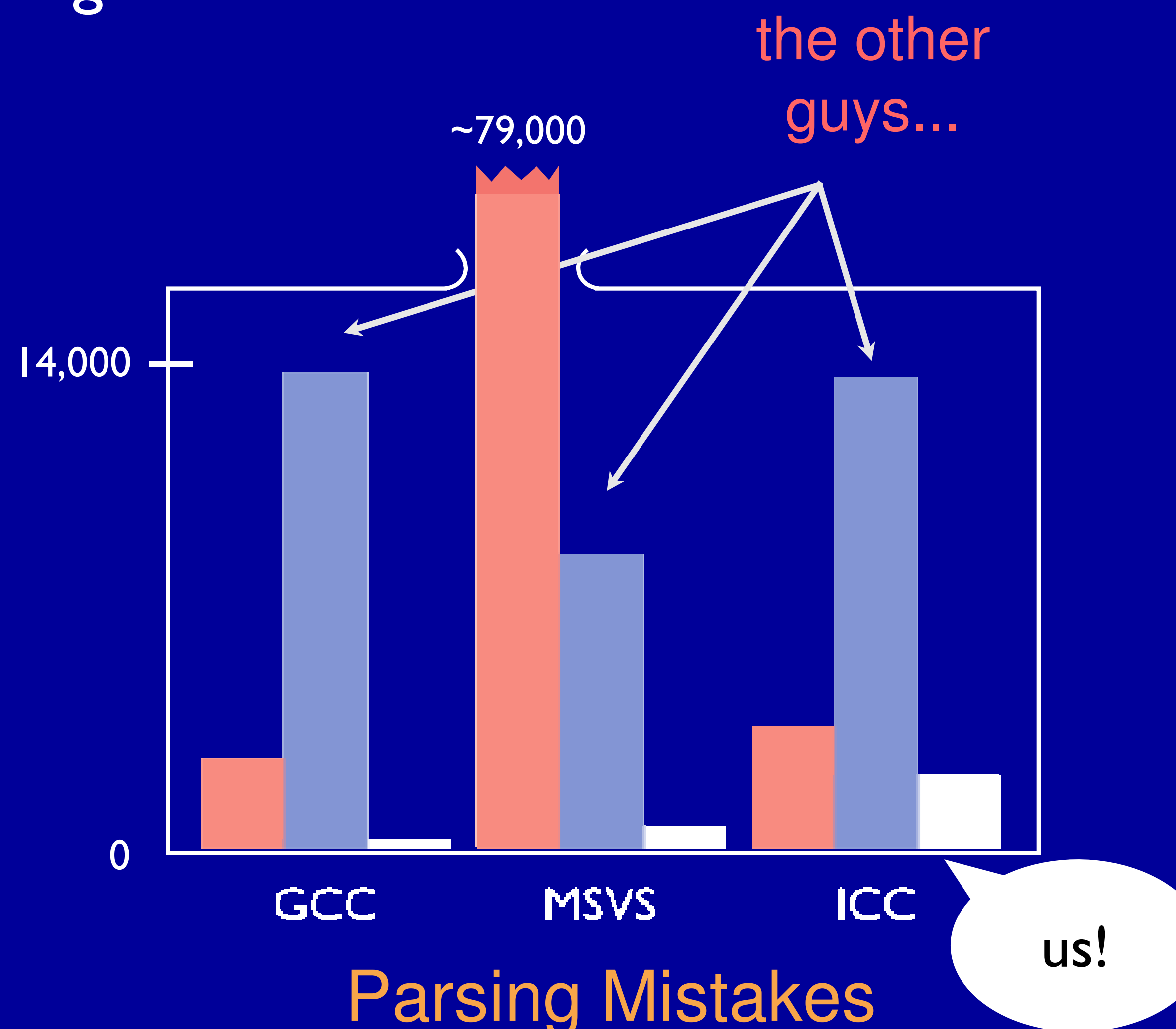
So... we're done, right?

Approach 1: Dynamic parsing at runtime

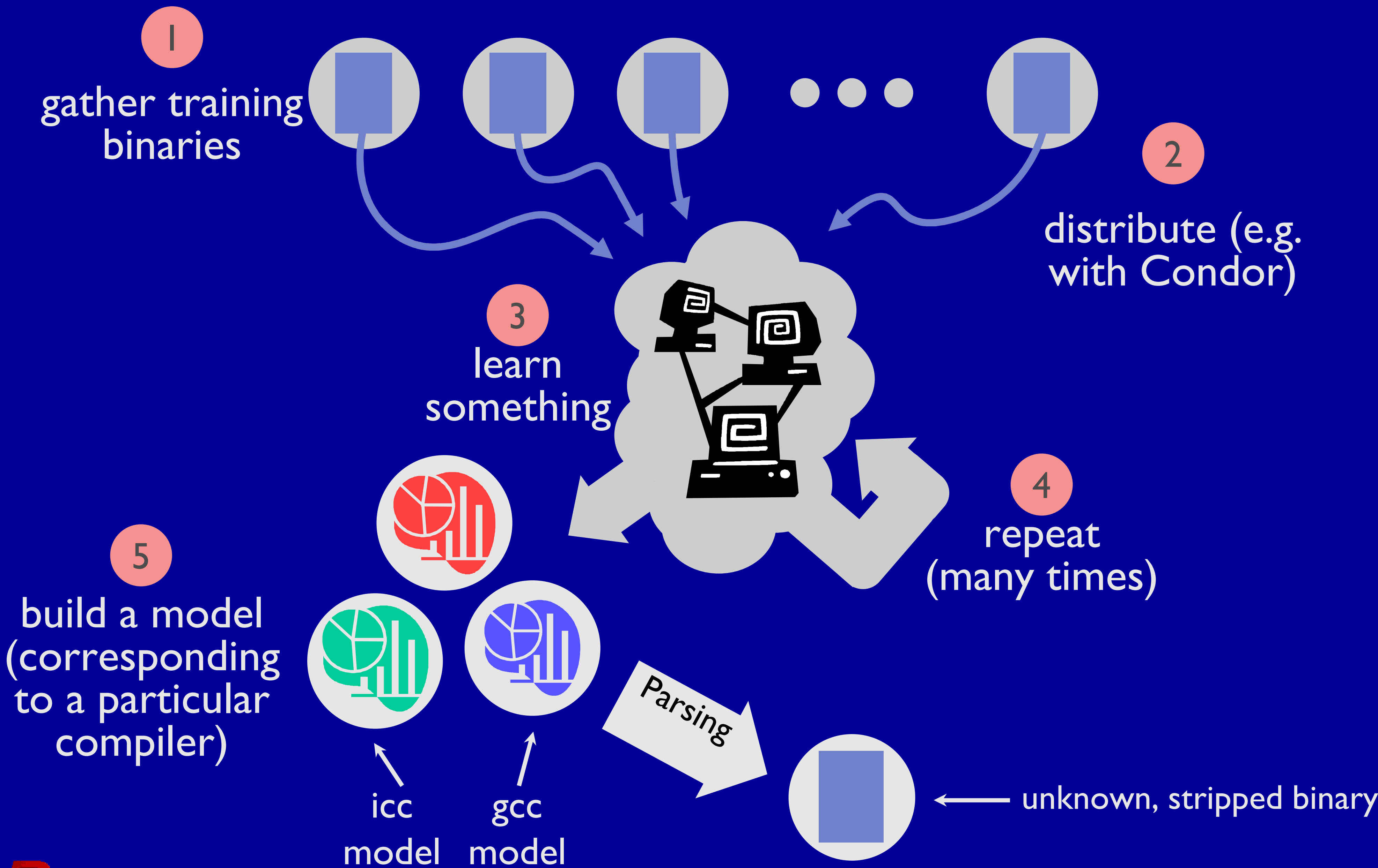
Appproach 2: Make static parsing better

Model many programs



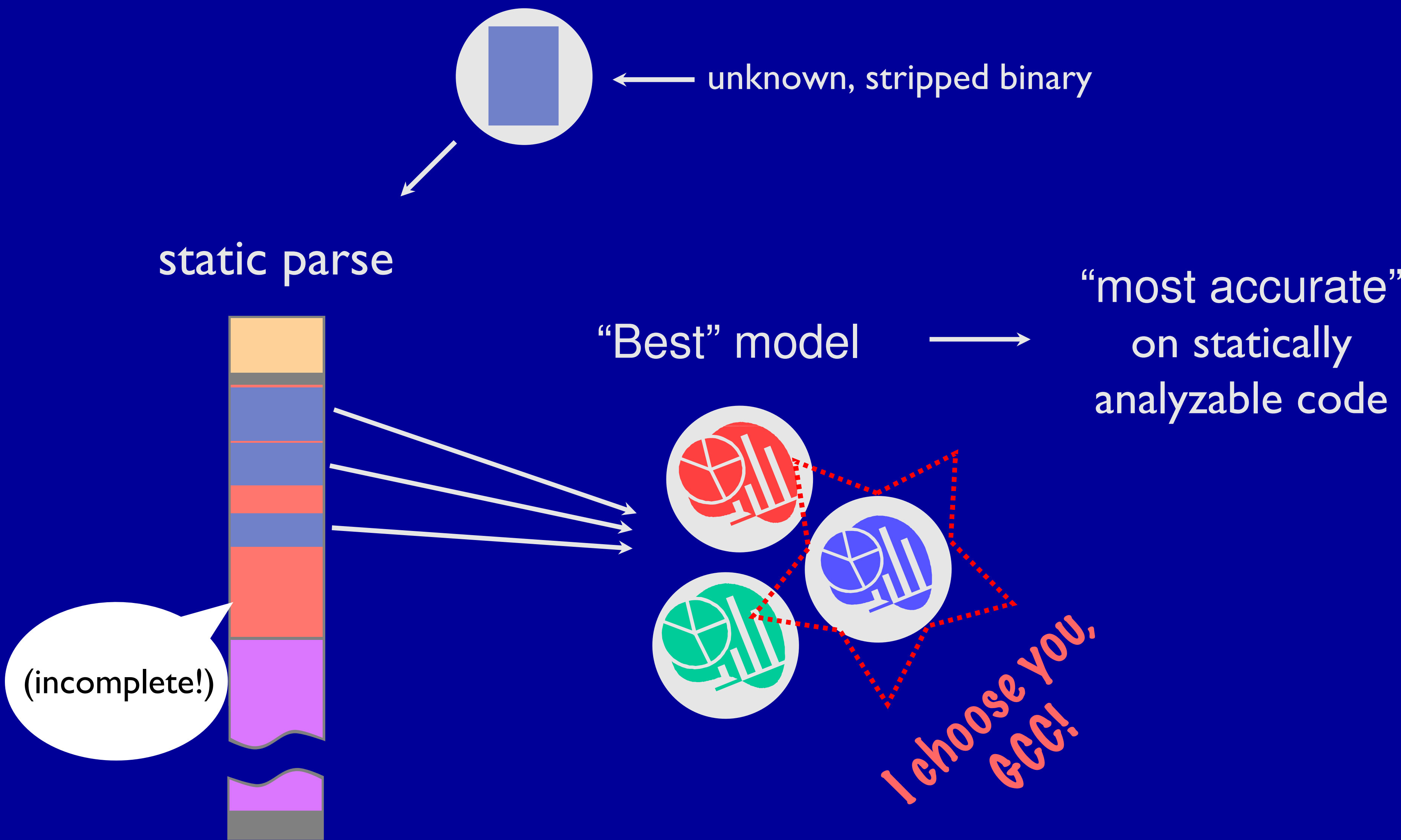


Looking for hidden assumptions

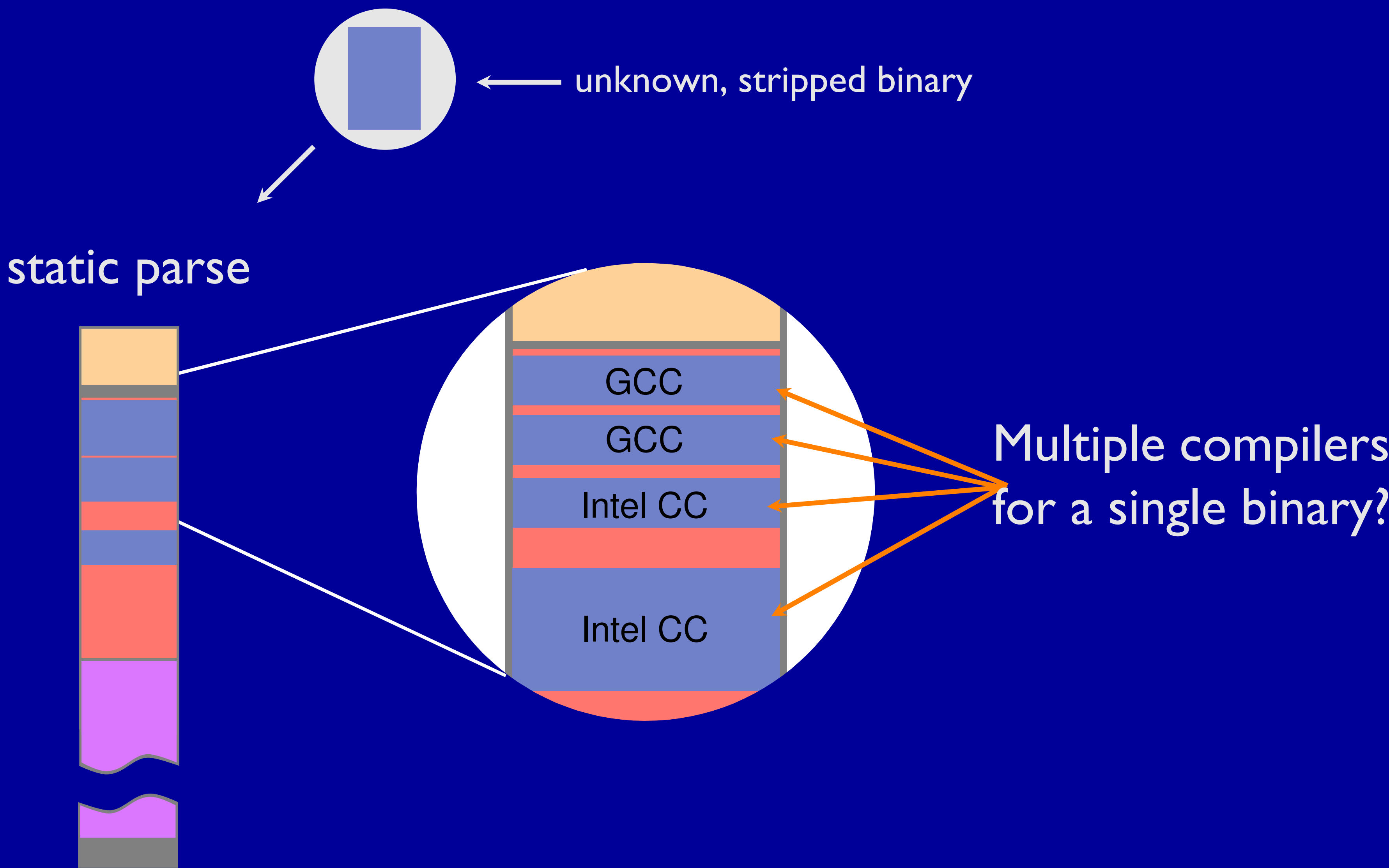



Wait, different
models per
compiler?

Reverse the problem



Simple solutions sometimes surprise

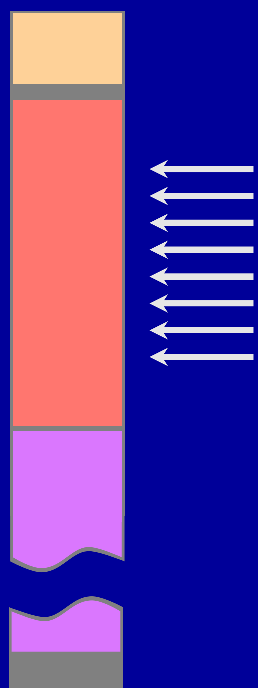


Not an exaggeration

Ex: Binary compiled by ICC with 2 GCC-compiled library methods

Side trip: alignment and conflict

stripped binary



parsing from every possible point

	candidate 1	candidate 2	candidate 3
14	add 0x14, esp		
53		push ebx	
83	sub 0xc, esp	sub 0xc, esp	
ec			
0c			or 0x8b, al
8b	mov 0x14(esp), ebx	mov 0x14(esp), ebx	
5c			pop esp
24			and 0x14, al
14			
8b	mov 0x4(ebx), edx	mov 0x4(ebx), edx	mov 0x4(ebx), edx
53			
4			
8b	mov 0x0(ebx), ecx	mov 0x0(ebx), ecx	mov 0x0(ebx), ecx
0b			

Model review: Idioms

Short sequences of instructions, possibly with wildcards, around function entry points

<< push ebp; mov esp,ebp >>

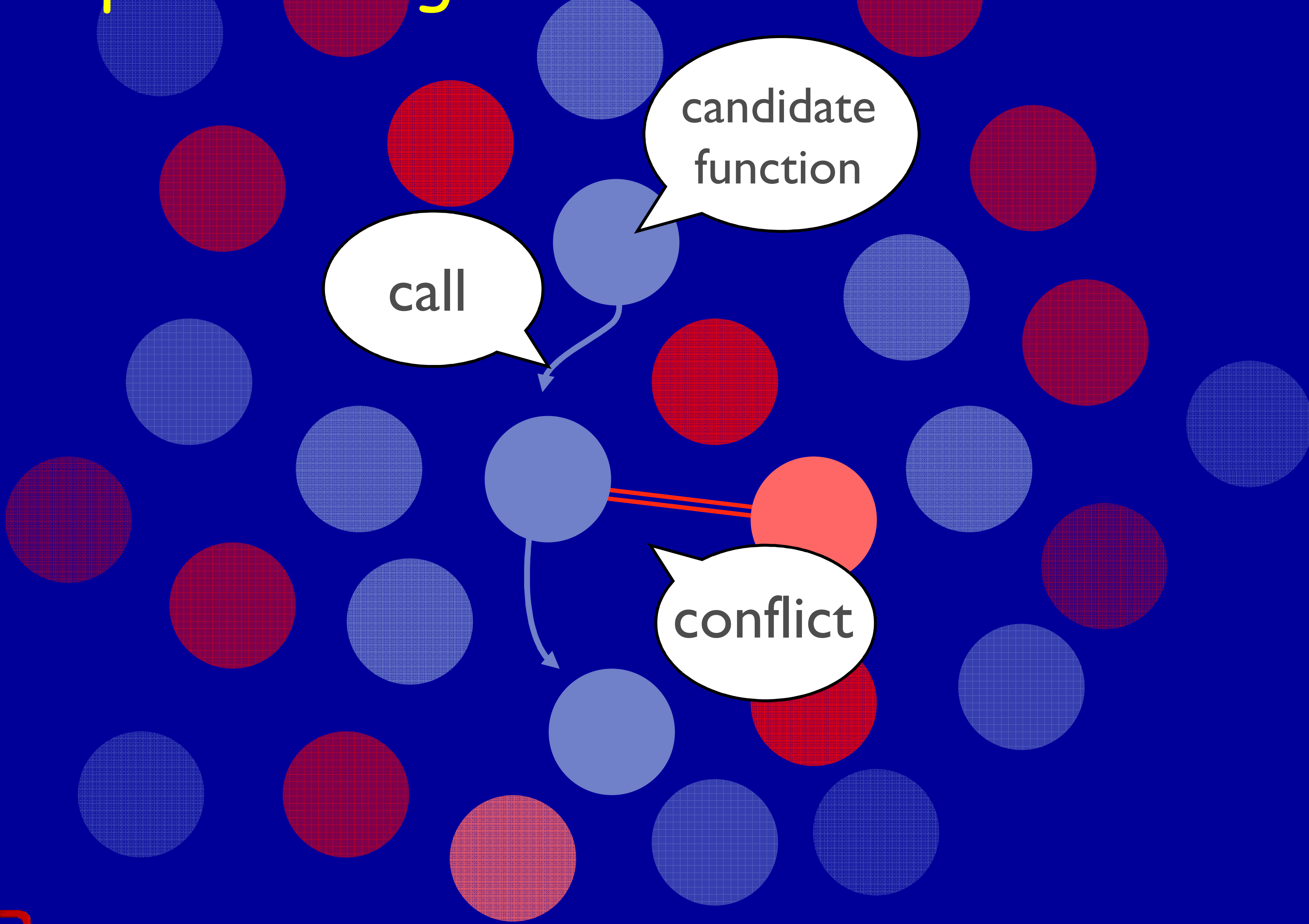
<< push ebp ; * ; mov esp,ebp >>

<< * ; mov 0x8(ebp),eax >>

PRE << ret ; nop >>

PRE idioms precede function entry point

Representing structure



Intractable!

labeling all possible functions *at once*

Probability of a labeling of "functions" over an n-byte binary

$$P(y_{1:n}|x_{1:n},\mathcal{P}) = \frac{1}{Z}\exp\left(\begin{array}{ll}\sum_{i=1}^{n}\sum_{u\in\{I\}}\lambda_u f_u(x_i,y_i,\mathcal{P}) & + \\ \sum_{i,j=1}^{n}\lambda_c f_c(x_i,x_j,y_i,y_j,\mathcal{P}) & + \\ \sum_{i,j=1}^{n}\lambda_o f_o(x_i,x_j,y_i,y_j,\mathcal{P}) & \end{array}\right)$$

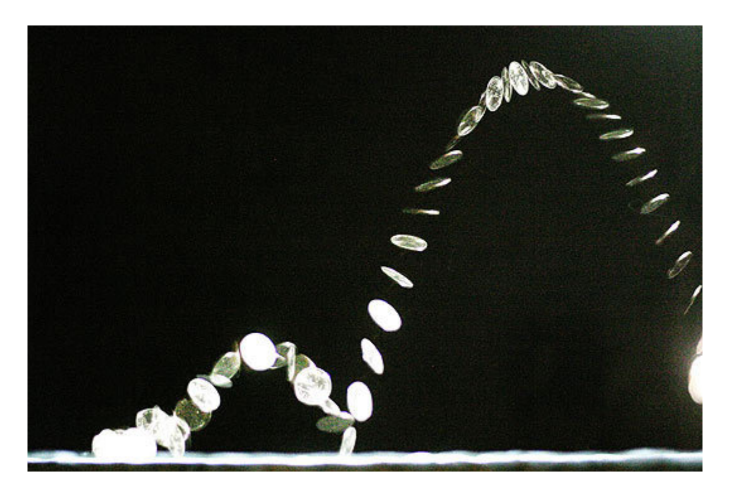
this guy is a killer!

Approximate!

Exact method → exponential in size of binary →

(heat death of universe)

Contrastive Divergence → very simple approximation →



(basically optimizing a related, easier problem)

(couple of coin flips)

Preliminary results are preliminary

1. Earlier successes are encouraging

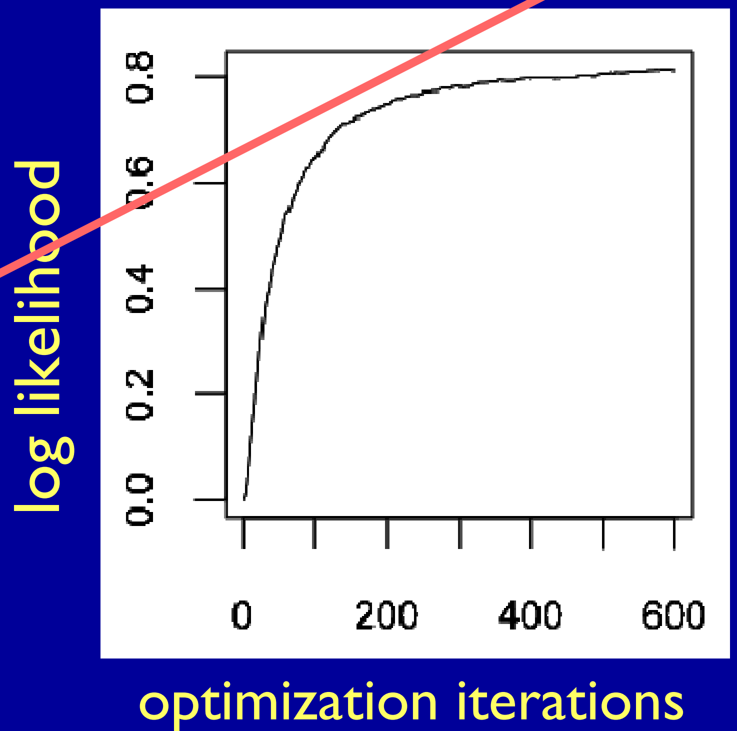
    10-fold experiment using "most accurate" model

    only 6 mistakes out of ~ 1000 programs

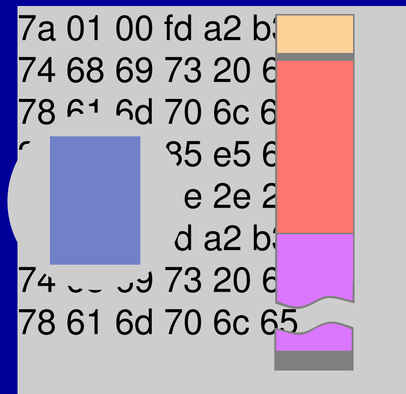
2. "Toy" results suggest approximate CD learning is effective

"small enough to compute exactly"

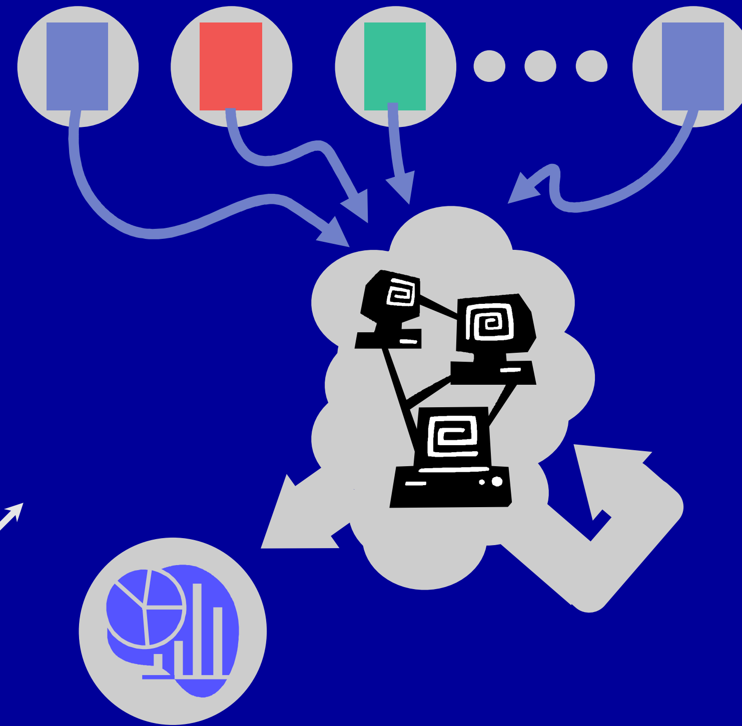
relatively smooth increase in likelihood during learning



Summing up

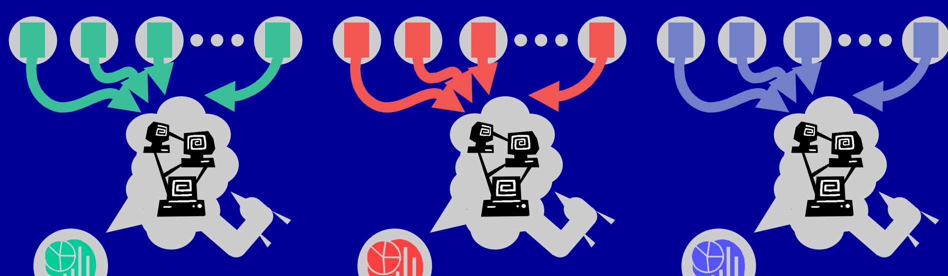


stripped binary parsing



general code models

function-granularity compiler identification!



compiler-specific code models

Backup slides follow

Experimentation

- GNU C Compiler
  - Simple, regular function preamble
- MS Visual Studio
  - High variation in function entry point idioms
- Intel C Compiler
  - Most variation in entry points; highly optimized

Compiler	Programs examined	Total Training Examples (pos+neg)	Total Test Examples (pos+neg)	Actual number of functions in gaps
GCC	625	8,412,711	22,806,449	85,870
MS VS	443	8,020,828	11,231,721	70,620
ICC	112	1,364,598	13,169,487	47,841

Testing

Comparison of three binary analysis tools:

- Original Dynisnt
  - Scans for common entry preamble
- IDA Pro Disassembler
  - Scans for common entry preamble
  - List of Library Fingerprints (Windows)
- Dyninst w/ Model
  - Model replaces entry preamble heuristic

Compiler	Orig. Dyninst		IDA Pro		Dyninst w/ Model	
	FP	FN	FP	FN	FP	FN
GCC	2,833	2,012	14,576	38,074	403	1,860
MS VS	79,320	65,586	9,044	21,491	725	14,143
ICC	3,786	40,195	14,422	26,970	2,337	16,220

Model Components