Labeling Library Functions in Stripped Binaries

Emily R. Jacobson, Nathan Rosenblum, and Barton P. Miller

Computer Sciences Department

University of Wisconsin - Madison

PASTE 2011

Szeged, Hungary

September 5, 2011

WISCONSIN

UNIVERSITY OF WISCONSIN–MADISON

Paradyn™

Why Binary Code?

- Source code isn't available

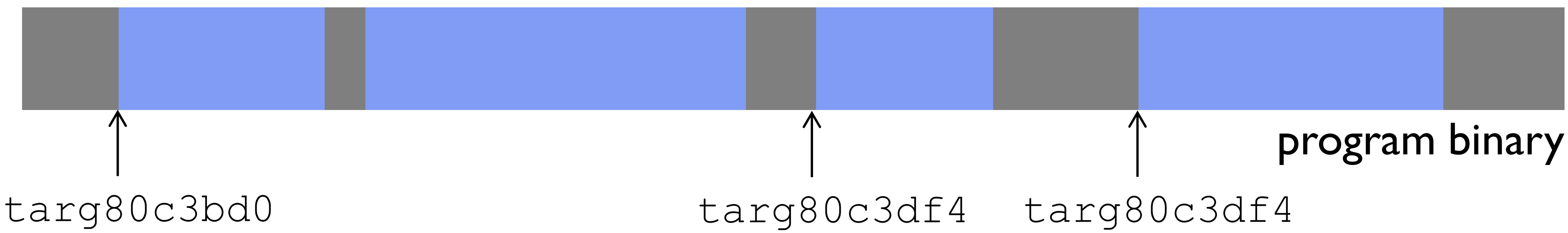
- Source code isn't the right representation

Binary Tools Need Symbol Tables

- Debugging Tools
  - GDB, IDA Pro…
- Instrumentation Tools
  - PIN, Dyninst,…
- Static Analysis Tools
  - CodeSurfer/x86,…
- Security Analysis Tools
  - IDA Pro,…

Restoring Information

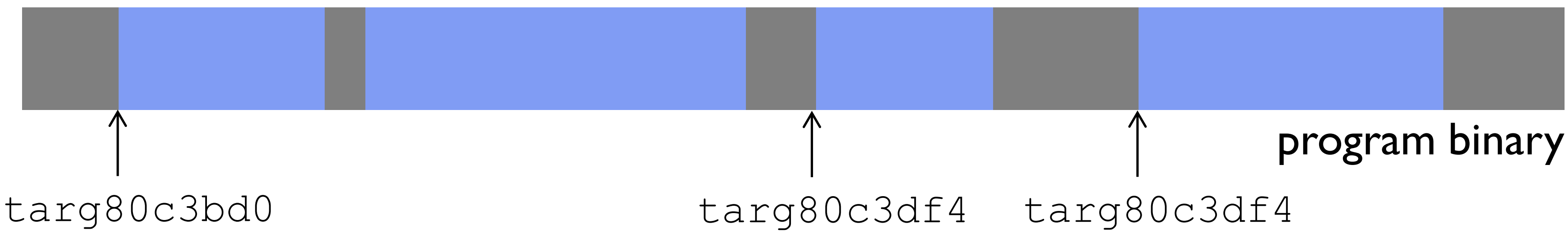
Function locations



Complicated by:

- Missing symbol information
- Variability in function layout (e.g. code sharing, outlined basic blocks)
- High degree of indirect control flow

Restoring Information



What about semantic information?

- Program's interaction with the operating system (*system calls*) encapsulated by *wrapper functions*

> *Library fingerprinting: identify functions based on patterns learned from exemplar libraries*

unstrip

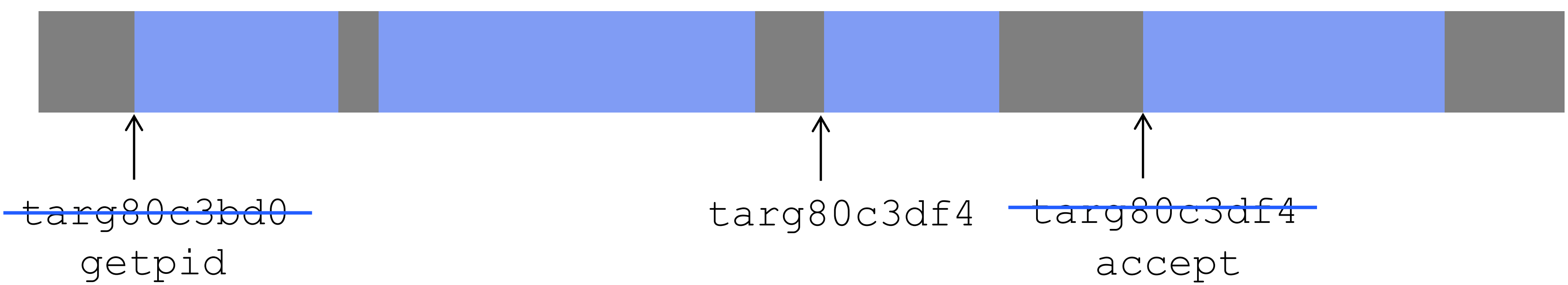
stripped binary parsing

+

library fingerprinting

+

binary rewriting



Save registers

Invoke a system call

```
<accept>:
    mov %ebx, %edx
    mov %0x66,%eax
    mov $0x5,%ebx
    lea 0x4(%esp),%ecx
    int $0x80
    mov %edx, %ebx
    cmp %0xffffff83,%eax
    jae syscall_error
    ret
```

Set up system call arguments

Error check and return

```
<accept>:
    mov %e
    mov %0
    mov $0
    0x
    int $0x80
    mov %edx, %ebx
    3,%eax
    ror
```

glibc 2.2.4 on RHEL with GCC 2.95.3

```
<accept>:
    cmpl $0x0,%gs:0xc
    jne 80f669c
    mov %ebx, %edx
    mov %0x66,%eax
    mov $0x5,%ebx
    lea 0x4(%esp),%ecx
    int $0x80
    mov %edx, %ebx
    cmp %0xffffff83,%eax
    jae syscall_error
    ret
    push %esi
    call enable_asyncancel
    mov %eax,%esi
    mov %ebx,%edx
    mov $0x66,%eax
    mov $0x5,%ebx
    lea 0x8(%esp),%ecx
    int $0x80
    mov %edx, %ebx
    xchg %eax,%esi
    call disable_acynancel
    mov %esi,%eax
    pop %esi
    cmp $0xffffff83,%eax
    jae syscall_error
    ret
```

glibc 2.5 on RHEL with GCC 4.1.2

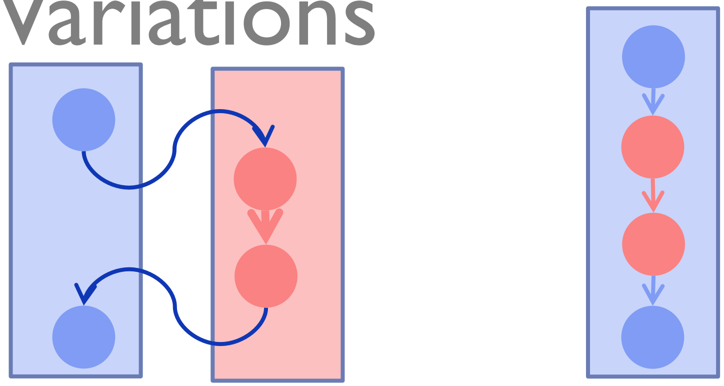
```
<accept>:
    cmpl $0x0,%gs:0xc
    jne 80f669c
    mov %ebx, %edx
    mov %0x66,%eax
    mov $0x5,%ebx
    lea 0x4(%esp),%ecx
    call *0x814e93c
    mov %edx, %ebx
    cmp %0xffffff83,%eax
    jae syscall_error
    ret
    push %esi
    call enable_asyncancel
    mov %eax,%esi
    mov %ebx,%edx
    mov $0x66,%eax
    mov $0x5,%ebx
    lea 0x8(%esp),%ecx
    call *0x8181578
    mov %edx, %ebx
    xchg %eax,%esi
    call disable_acynancel
    mov %esi,%eax
    pop %esi
    cmp $0xffffff83,%eax
    jae syscall_error
    ret
```

glibc 2.5 on RHEL with GCC 3.4.4

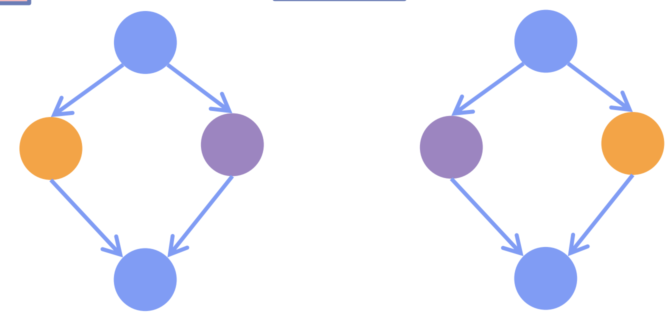
The same function can be realized in a variety of ways in the binary

Binary-level Code Variations

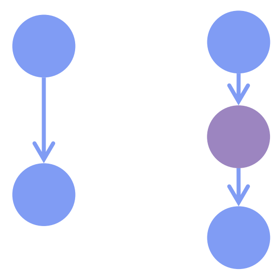
- Function inlining



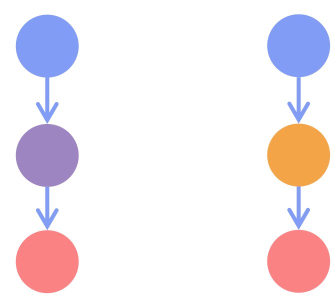
- Code reordering



- Minor code changes



- Alternative code sequences



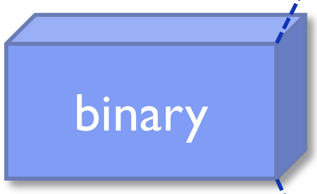
Semantic Descriptors

- Rather than recording byte patterns, we take a semantic approach
- Record information that is likely to be invariant across multiple versions of the function

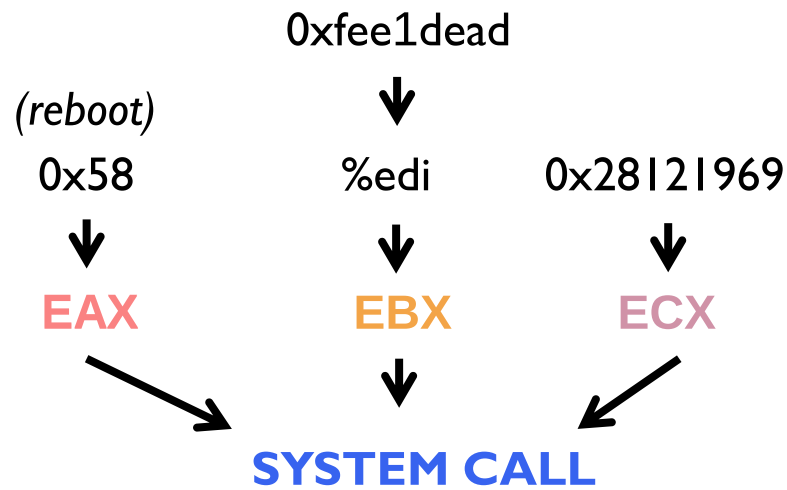
```
<accept>:
    mov %ebx, %edx
    mov %0x66,%eax
    mov $0x5,%ebx
    lea 0x4(%esp),%ecx
    int $0x80
    mov %edx, %ebx
    cmp %0xffffff83,%eax
    jae 8048300
    ret
    mov %esi,%esi
```

→ {<socketcall, 5 >}

Building Semantic Descriptors




```
reboot:
    push %ebp
    mov %esp,%ebp
    sub $0x10,%esp
    push %edi
    push %ebx
    mov 0x8(%ebp),%edx
    mov $0xfee1dead,%edi
    mov $0x28121969,%ecx
    push %ebx
    mov %edi,%ebx
    mov $0x58,%eax
    int $0x80
    …
```



{<reboot, 0xfee1dead, 0x2812969>}

We parse an input binary, locate system calls and wrapper function calls, and employ dataflow analysis.

Building Semantic Descriptors Recursively

```
open:
    …
    mov $0x5, eax
    int $0x80
    …
```

{<open, “/etc/hostid”, 577, 420>}

```
sethostid:
    …
    call open
    …
    call write
    …
    mov $0x6, eax
    int $0x80
```

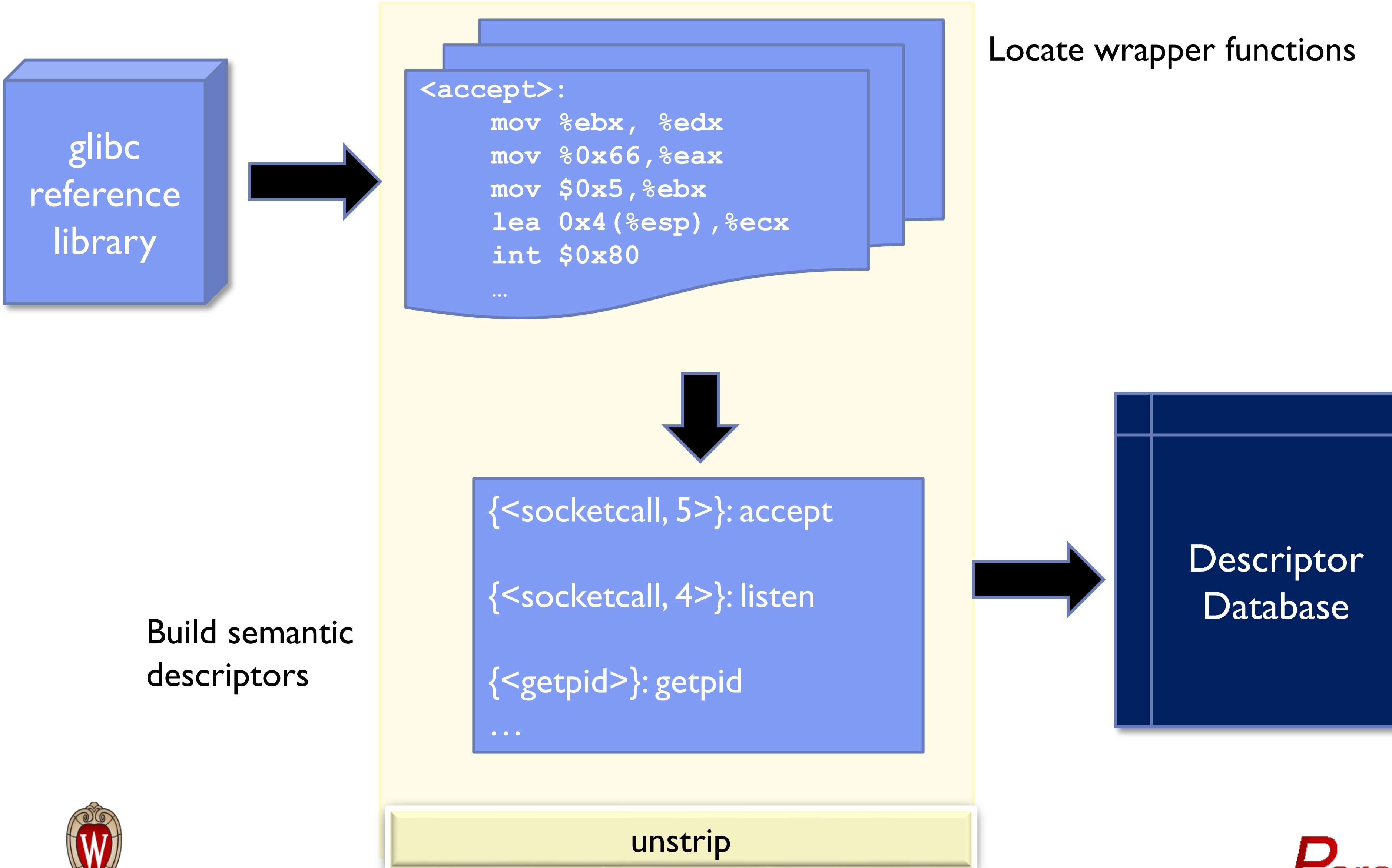
~~{<close>}~~

```
write:
    …
    mov $0x4, eax
    int $0x80
    …
```

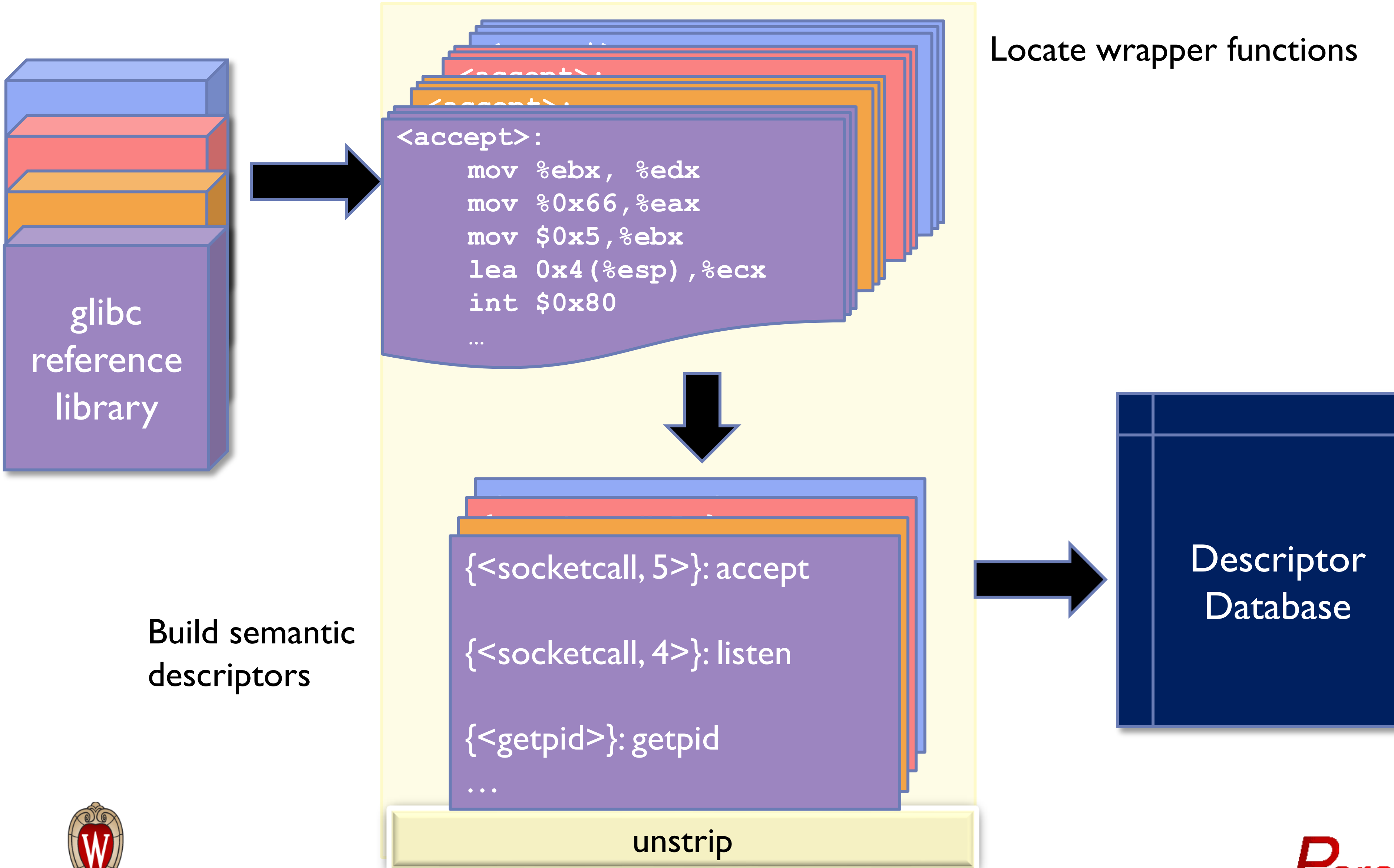
{<write,?,?,4>}

{ <close>, <open, “/etc/hostid”, 577,420>, <write,?,?,4>}

Building a Descriptor Database



Building a Descriptor Database



Pattern Matching Criteria

- Two stages
  1) Exact matches
  2) Best match based on coverage criterion

- Handle minor code variations by allowing flexible matches

Pattern Matching Criteria

fingerprint from the **database**

↓

A: {<socketcall,5>}

B: {<socketcall,5>, <socketcall,5>, <futex>}

↑

semantic descriptor from the **code**

$$coverage(A,B) = \frac{|A \cap B|}{|B|}$$

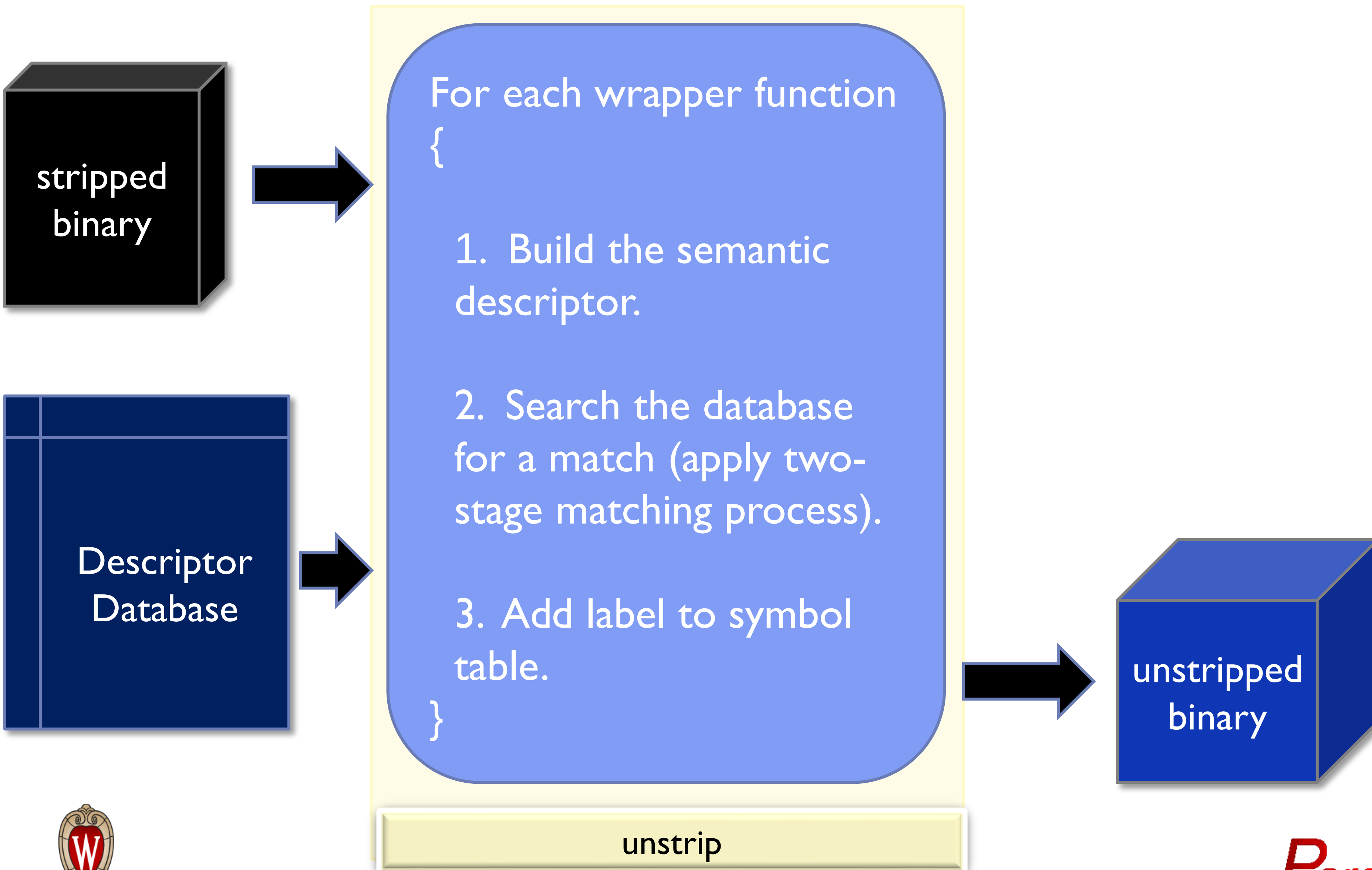
$$A \cap B = \{ b \in B \mid b \in A \}$$

$$coverage(A,B) = \frac{2}{3}$$

Multiple Matches

- It's possible that two or more functions are indistinguishable
- Policy decision: return set of potential matches
- In practice, we've observed 8% of functions have multiple matches, but the size of the match set is small (≤ 3)

Identifying Functions in a Stripped Binary



Implementation

Dyninst

stripped binary parsing

ParseAPI
A Dyninst Component

+

library fingerprinting

InstructionAPI
A Dyninst Component

DataflowAPI
A Dyninst Component

+

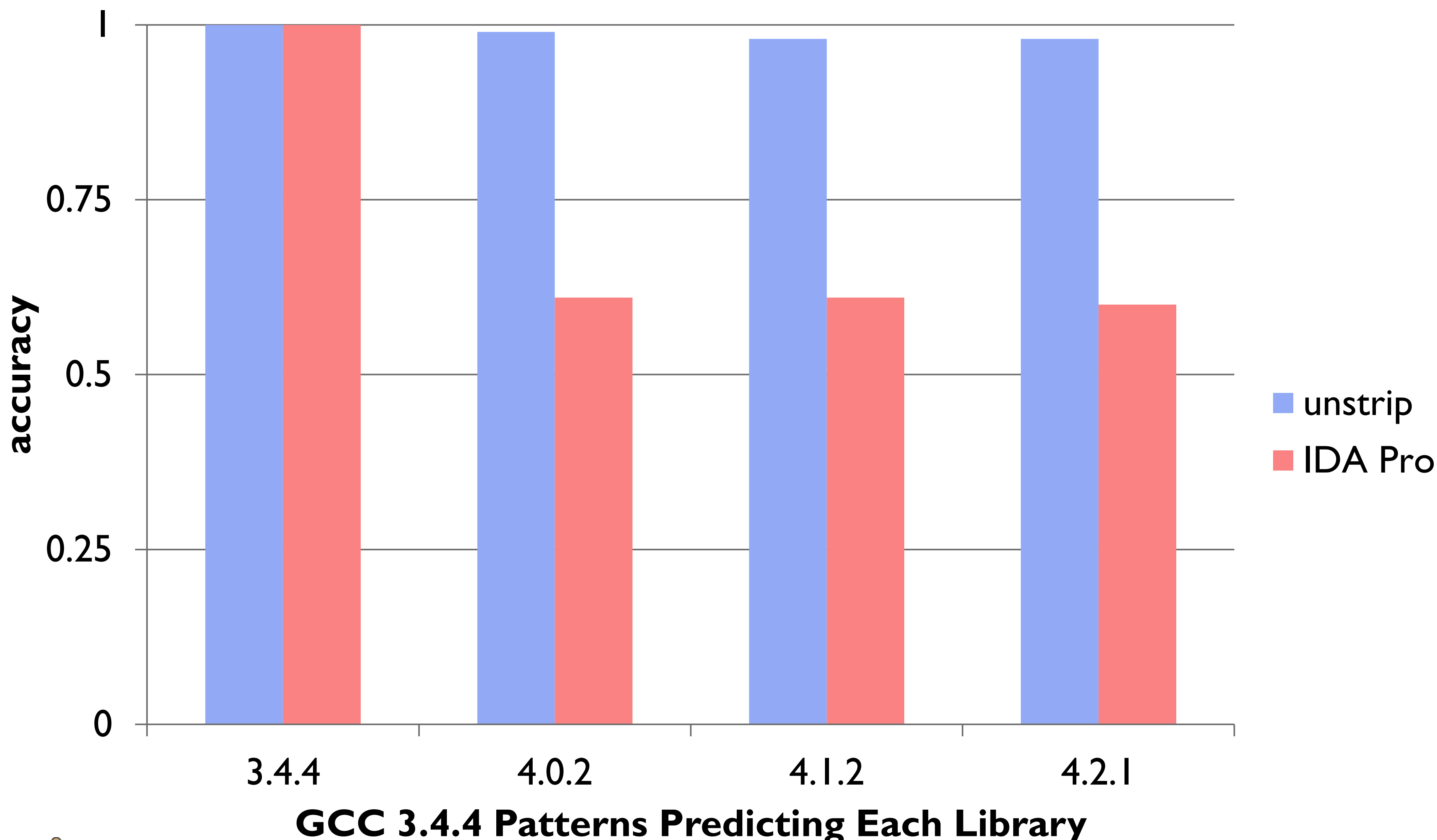
binary rewriting

SymtabAPI
A Dyninst Component

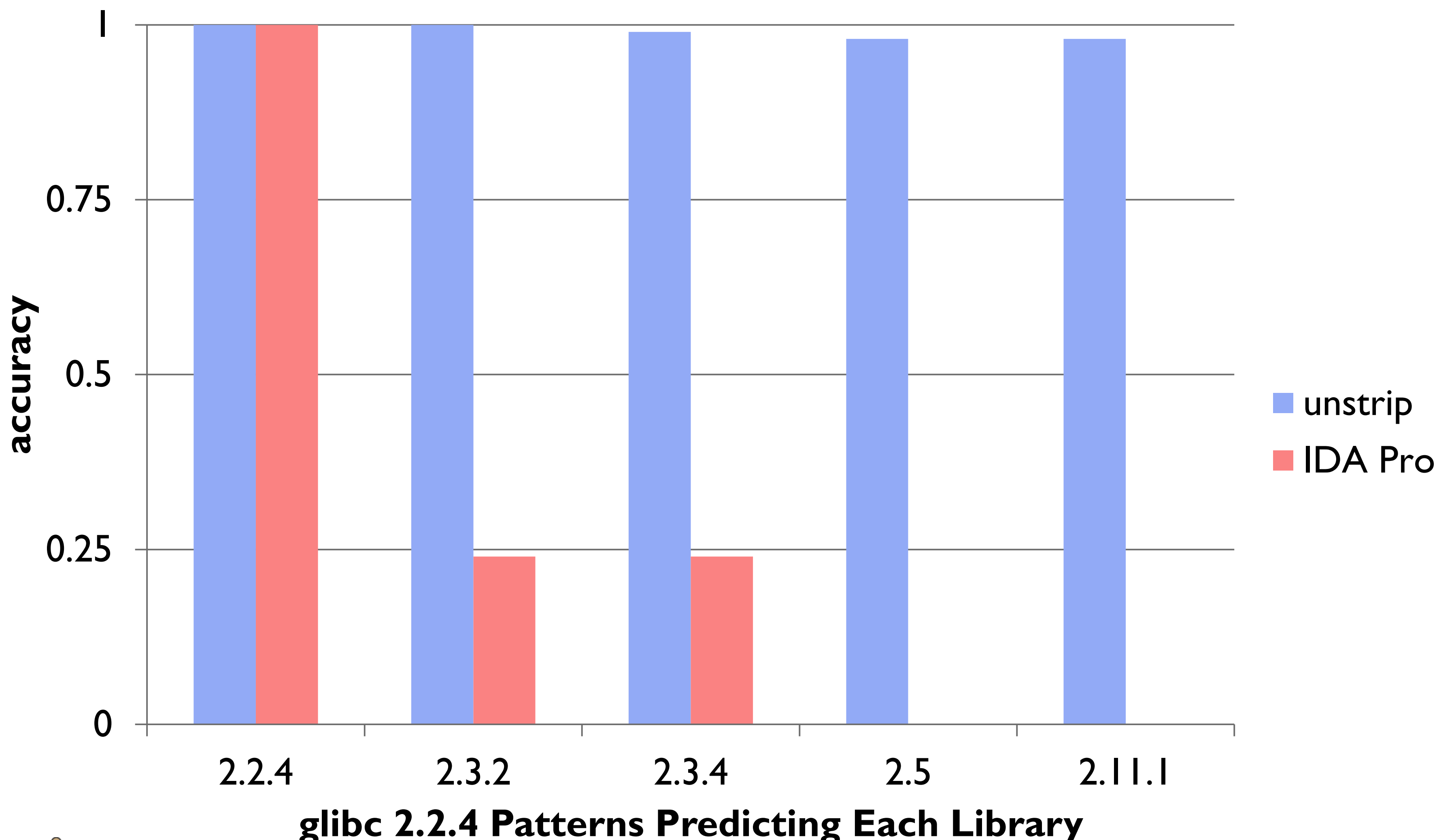
Evaluation

- To evaluate across three dimensions of variation, we constructed three data sets:
  - GCC version
  - glibc version
  - distribution vendor
- In each set, compile statically-linked binaries, build a DDB, compare unstrip to IDA Pro's FLIRT
- Evaluation measure is accuracy

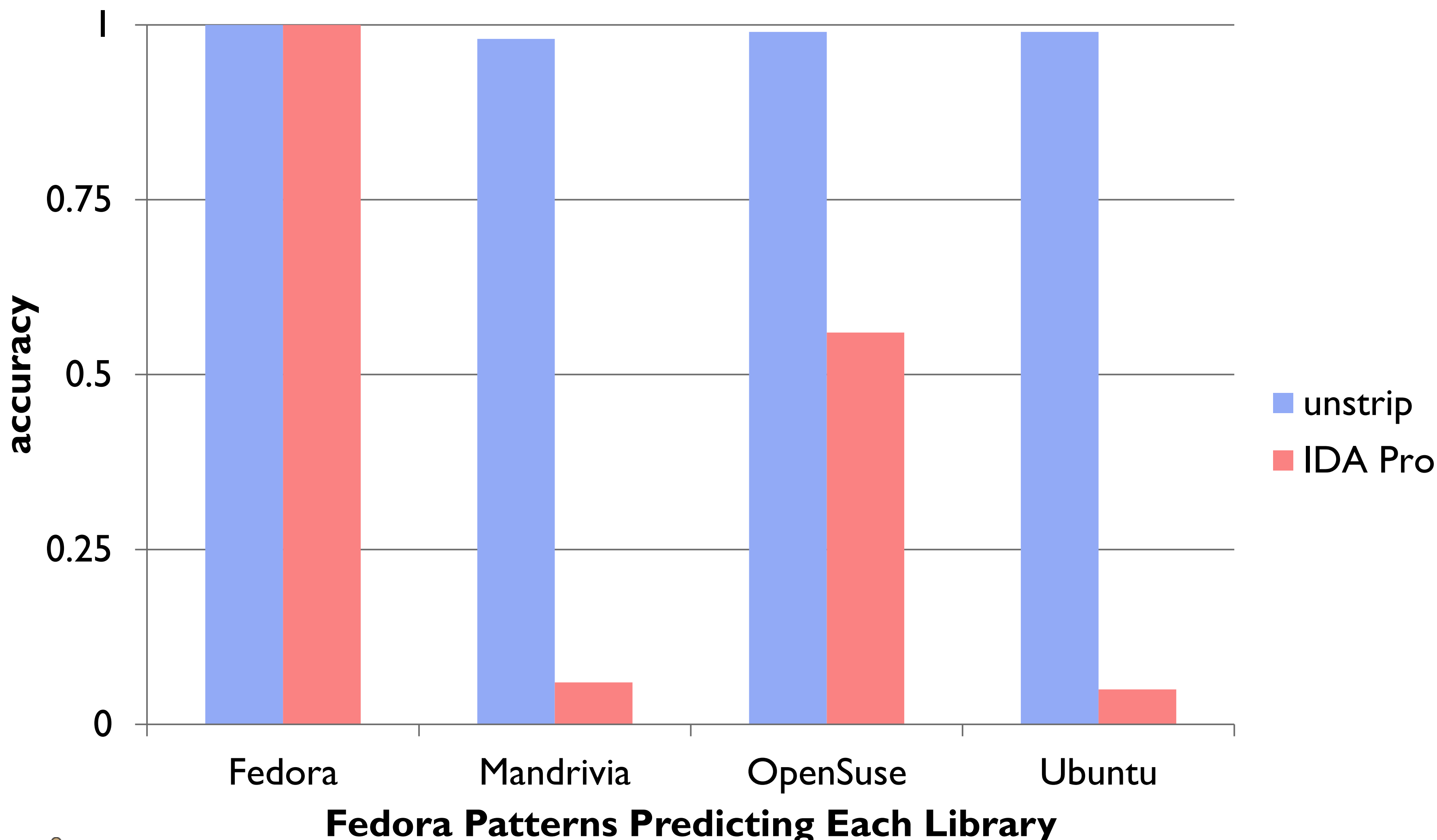
Evaluation Results: GCC Version Study



Evaluation Results: glibc Version Study



Evaluation Results: Distribution Study

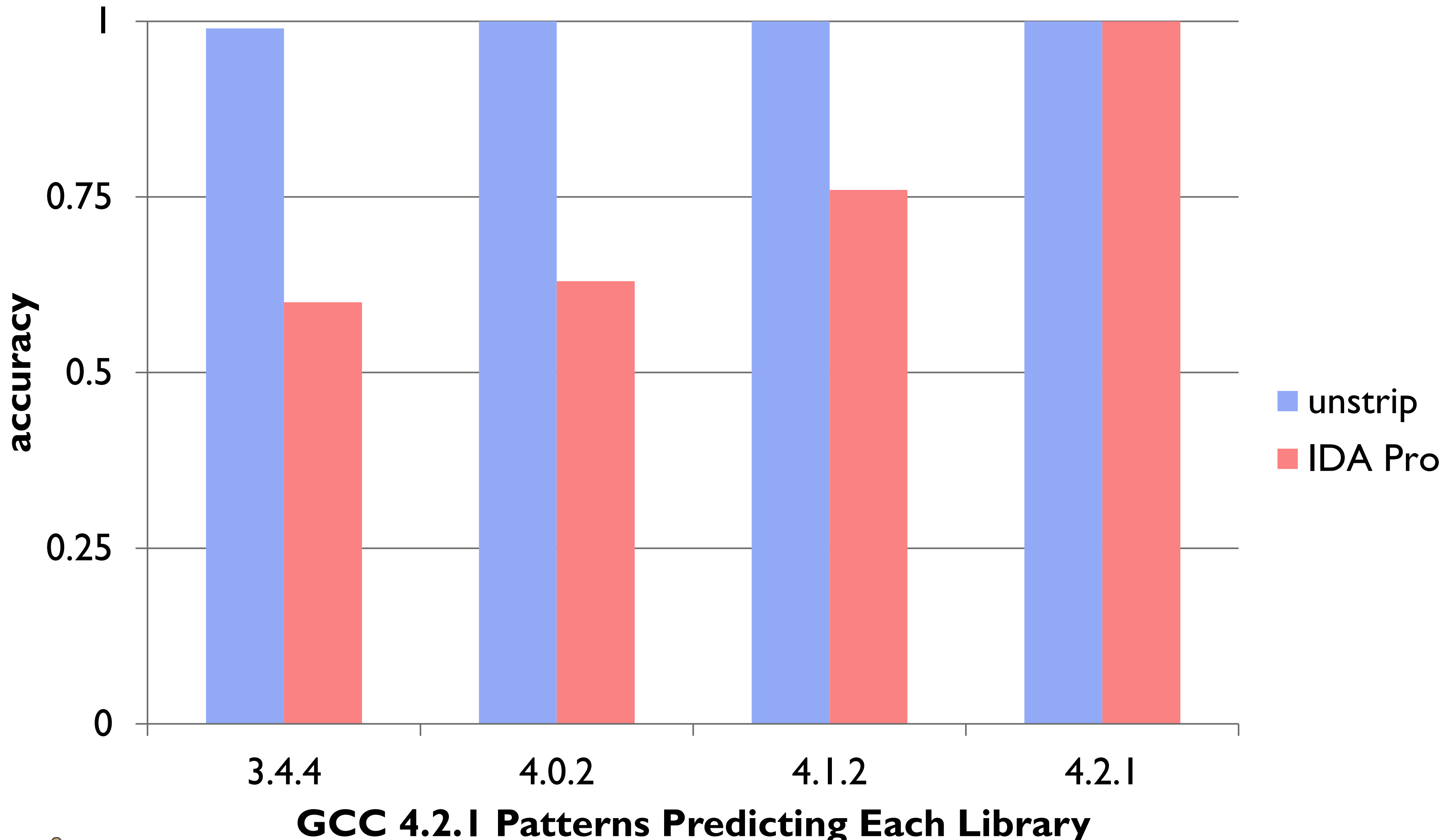


unstrip is available at

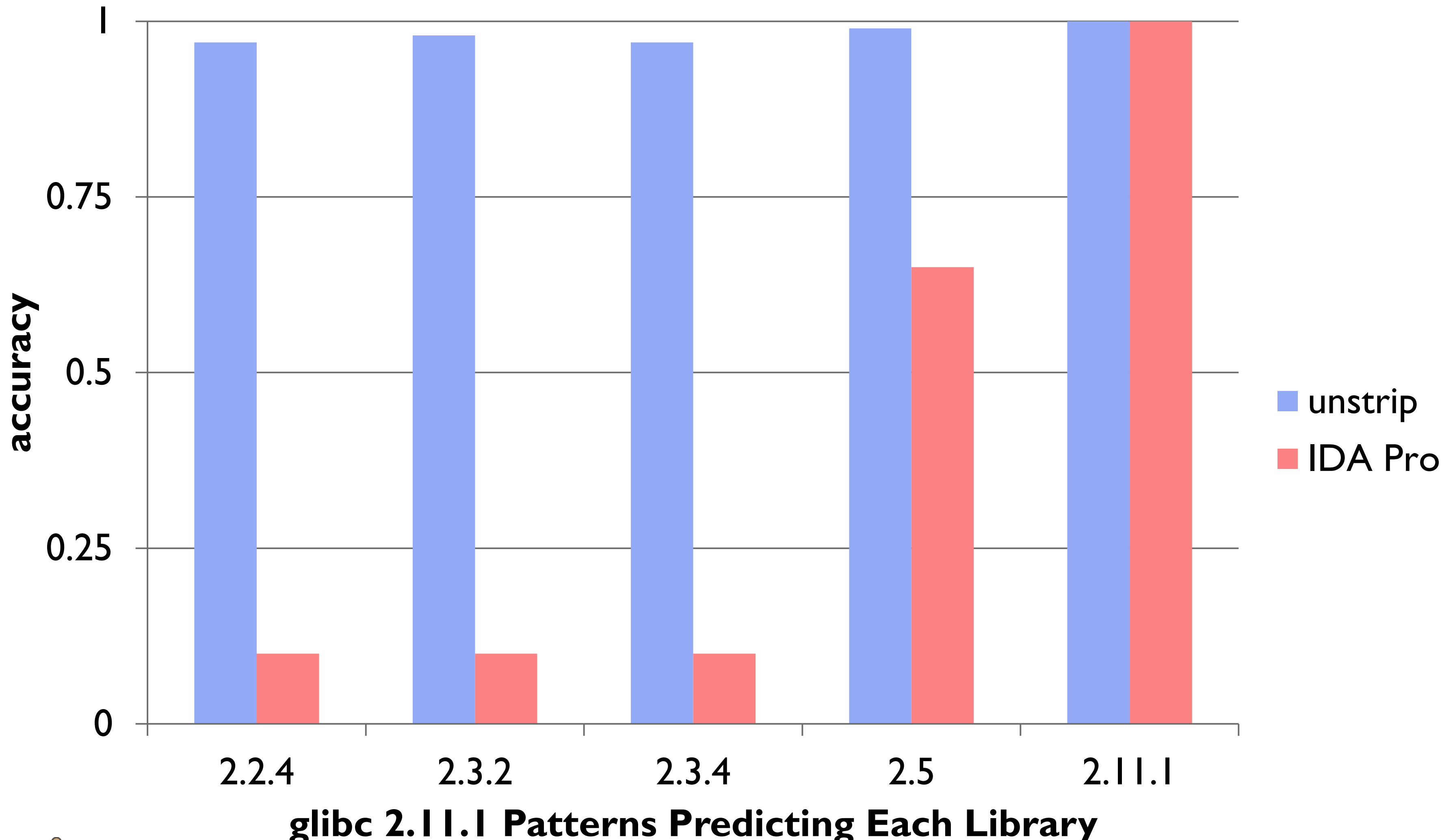
http://www.paradyn.org/html/tools/unstrip.html

Backup slides follow

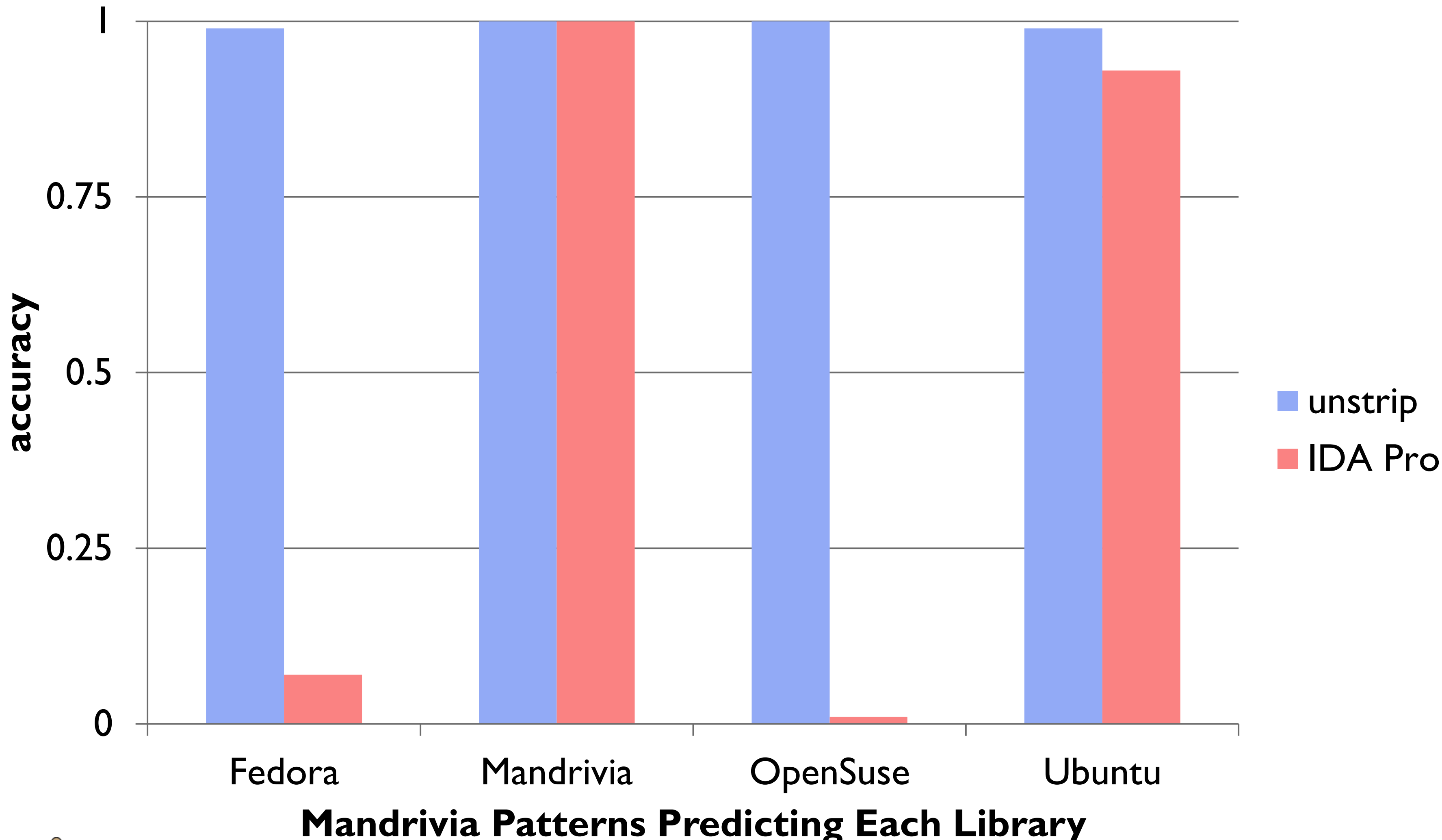
Evaluation Results: GCC Version Study (Temporal: backwards)



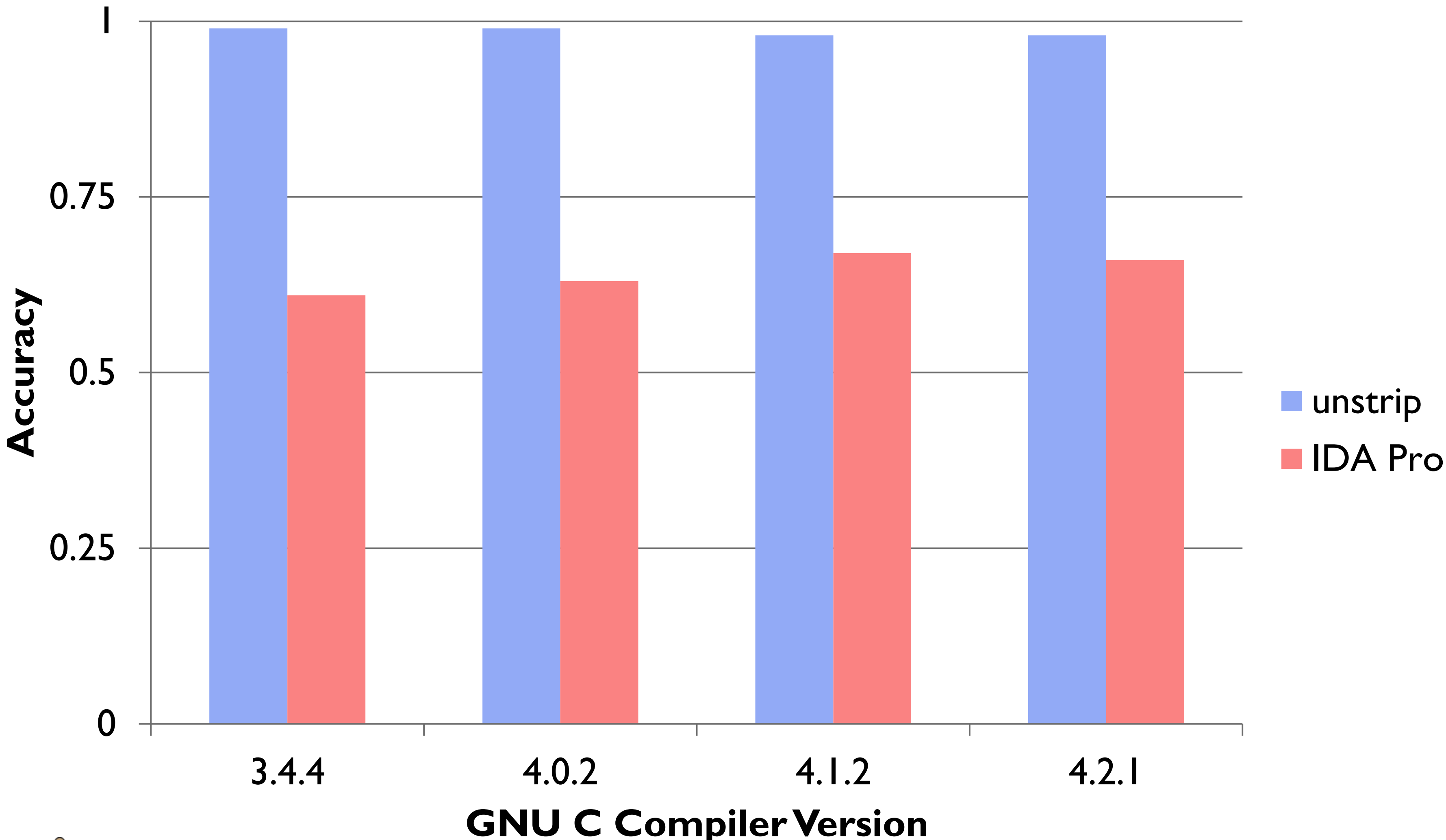
Evaluation Results: glibc Version Study (Temporal: backwards)



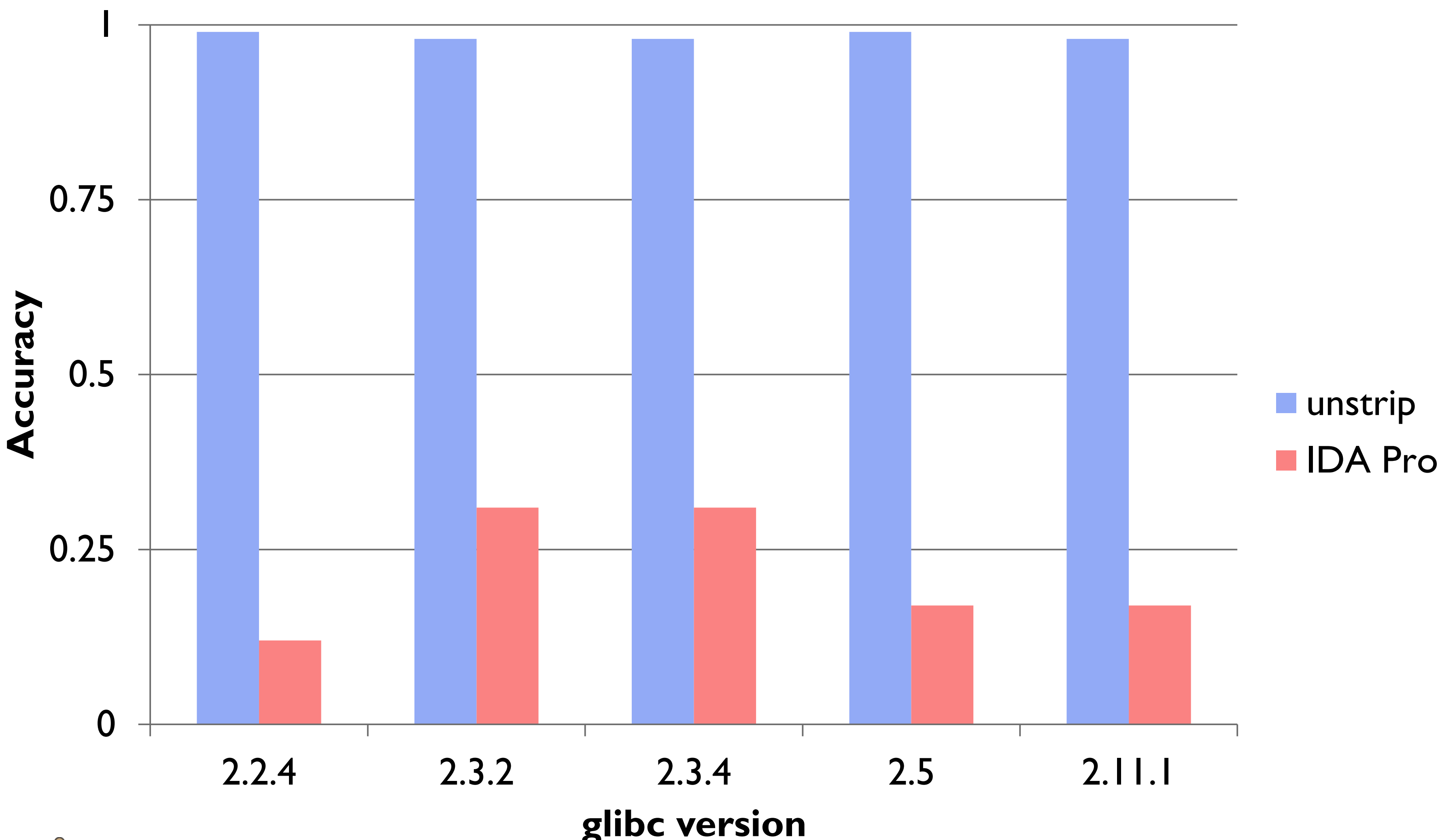
Evaluation Results: Distribution Study (one predicts the rest)



Evaluation Results: GCC Version Study (one predicts the rest)



Evaluation Results: glibc Version Study (one predicts the rest)



Evaluation Results: Distribution Study (one predicts the rest)