

# String Analysis for Binaries

Mihai Christodorescu

Nicholas Kidd

Wen-Han Goh

University of Wisconsin, Madison

# What is *String Analysis*?

- Recovery of values a string variable might take at a given program point.

```
void main( void )
{
    char * msg = "no msg";
    ▶ printf( "This food has %s.\n", msg );
}
```

**Output:** This food has no msg.

# Why Do We Need String Analysis?

- We could just use the `strings` program:

```
$ strings no_msg  
/lib/ld-linux.so.2  
libc.so.6
```

```
...
```

```
...
```

- ▶ `no msg`
  - ▶ `This food has %s.`
- ```
$
```

# Why Perform String Analysis?

- **Computer forensics**

Given an unknown program, we want to know the files it might access, the registry keys it might get and set, the commands it might execute.

- **Program verification**

SQL queries, embedded scripting, ...

# A Complicated Example

```
void main( void )
{
    char buf[257];
    strcpy( buf, "/" );
    strcat( buf, "b" );
    strcat( buf, "i" );
    strcat( buf, "n" );
    ...
    system( buf );
}
```

# A Complicated Example

```
void main( void )
{
    char buf[257];
    strcpy( buf, "/" );
    strcat( buf, "b" );
    strcat( buf, "i" );
    strcat( buf, "n" );
    ...
    system( buf );
}
```

Running strings:

```
/
a
b
c
d
...
```

# A Complicated Example

```
void main( void )
{
    char buf[257];
    strcpy( buf, "/" );
    strcat( buf, "b" );
    strcat( buf, "i" );
    strcat( buf, "n" );
    ...
    system( buf );
}
```

Running strings:

```
/
a
b
c
d
...
```

Running a string analysis:

```
/bin/ifconfig -a |
/bin/mail ...@...
```

# Our Contributions

- Developed a **string analysis for binaries**.
- Implemented **x86sa**, a string analyzer for Intel IA-32 binaries.
- Evaluated on both benign and malicious binaries.

# Outline

- String analysis for Java.
- String analysis for x86.
- Evaluation.
- Applications & future work.

# String Analysis for Java

*Christensen, Møller, Schwartzbach "Precise Analysis of String Expressions" (SAS'03)*

## 1. Create string flowgraph.

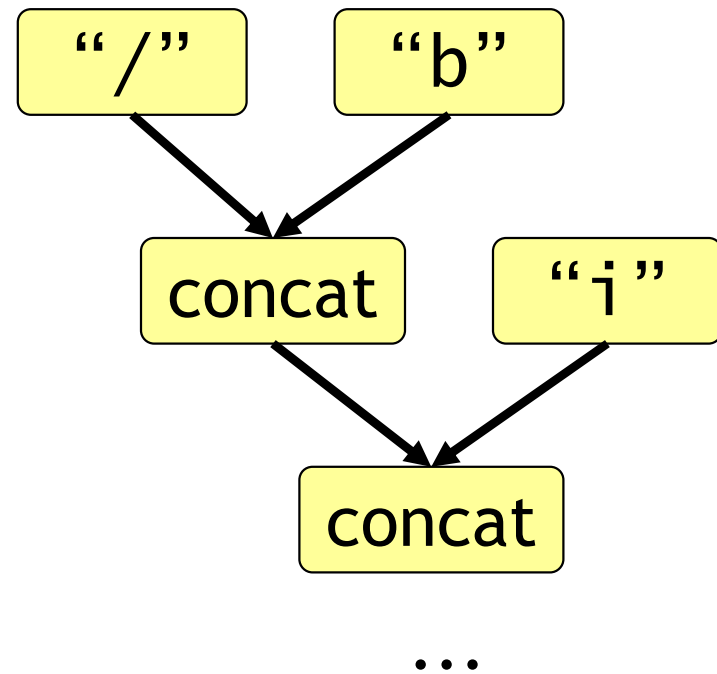
```
void main( void )
{
    String x = "/";
    x = x + "b";
    x = x + "i";
    x = x + "n";
    ...
    System.exec( x );
}
```

# String Analysis for Java

Christensen, Møller, Schwartzbach "Precise Analysis of String Expressions" (SAS'03)

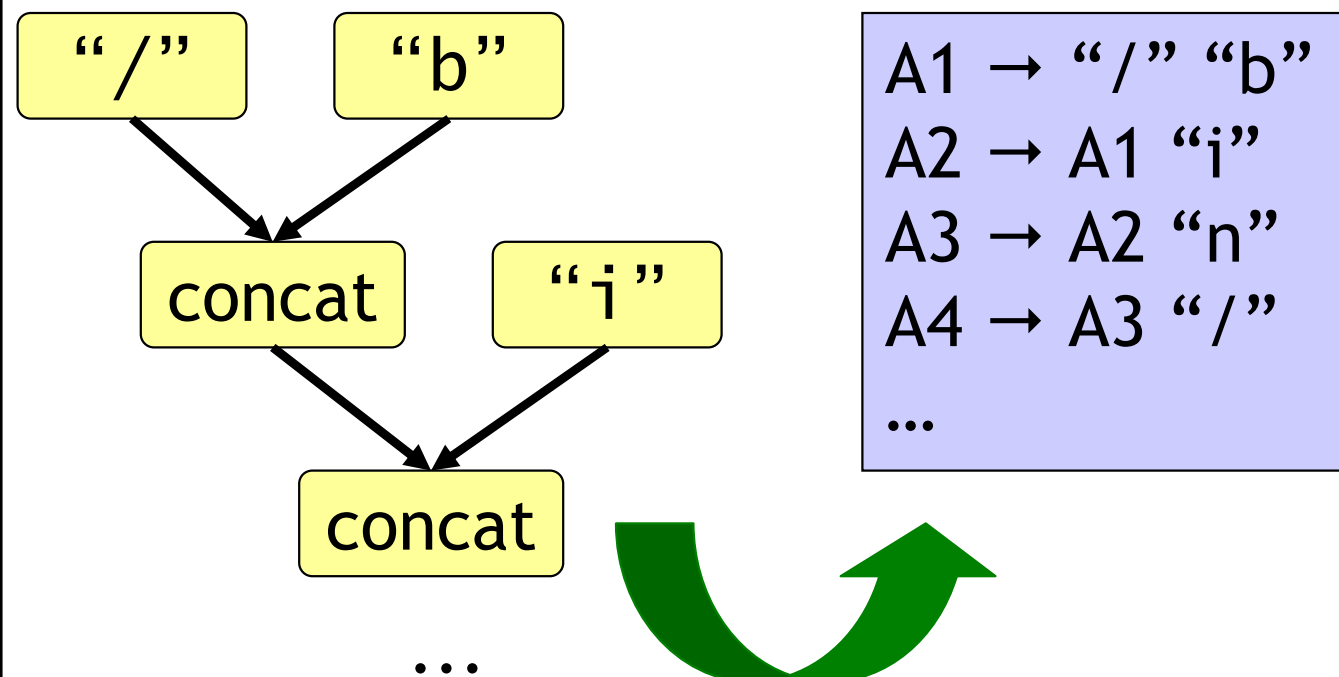
## 1. Create string flowgraph.

```
void main( void )  
{  
    String x = "/";  
    x = x + "b";  
    x = x + "i";  
    x = x + "n";  
    ...  
    System.exec( x );  
}
```



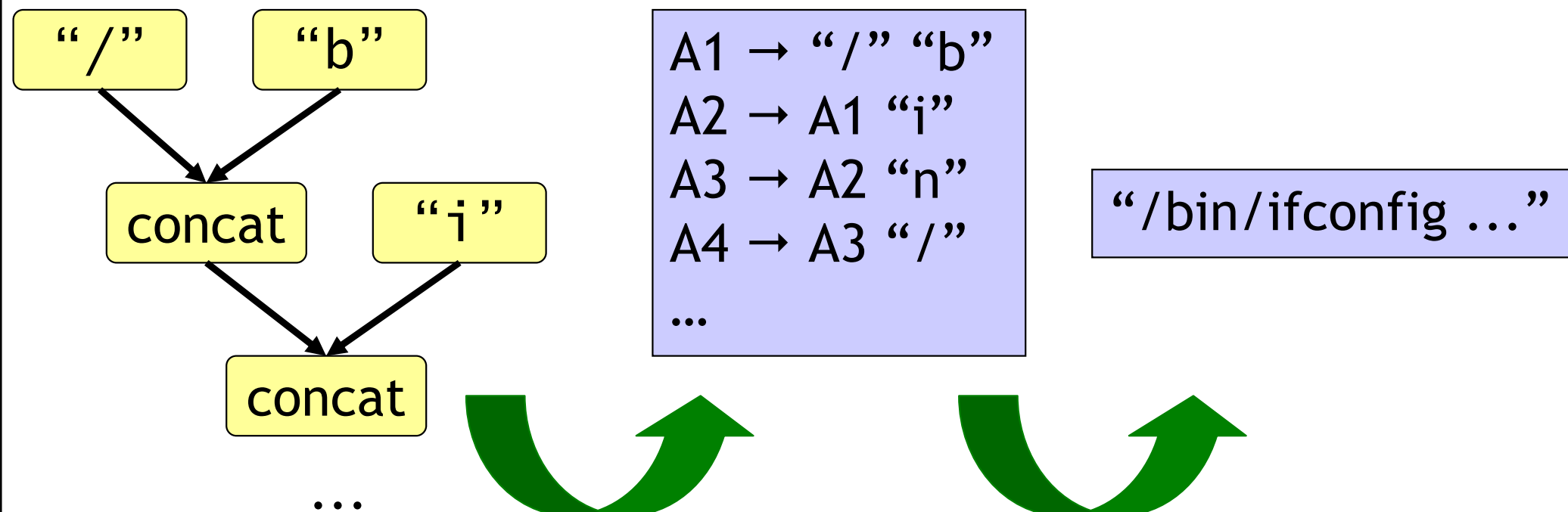
# String Analysis for Java [2]

## 2. Create context-free grammar.

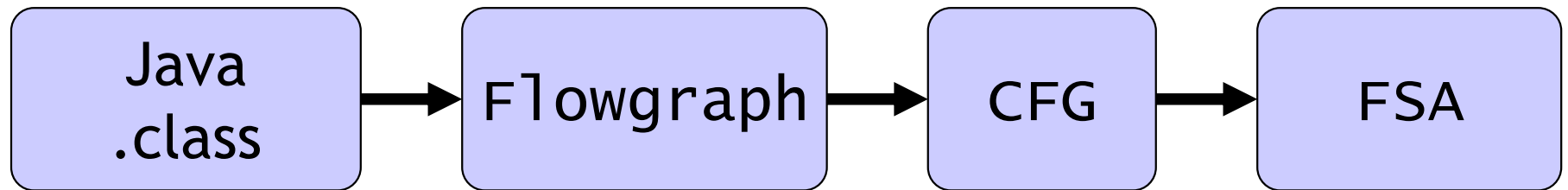


# String Analysis for Java [2]

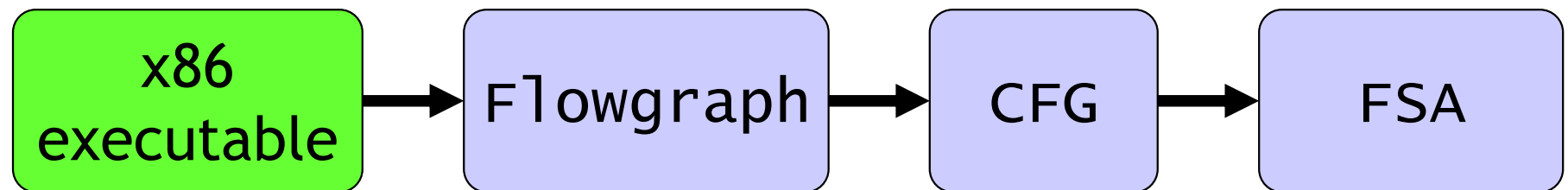
2. Create context-free grammar.
3. Approximate with finite automaton.



# From Java to x86 executables



# From Java to x86 executables



Rest of this talk:

Bridge the syntactic and semantic gaps between Java and assembly language.

# Outline

- String analysis for Java.
- String analysis for x86.
- Evaluation.
- Applications & future work.

# Four Problems with Assembly

1. No types.
2. No high-level constructs.
3. No argument passing convention.
4. No Java string semantics.

# Problem 1: No Types

- Solution: infer types from C lib. funcs.

## Assumption #1:

Strings are manipulated only using string library functions.

```
char * strcat( char * dest, char * src )
```

- After: “eax” points to a string.
- Before: “dest” and “src” point to a string.

# Problem 1: No Types [cont.]

- Perform a backwards analysis to find the strings:
  - Destination registers “kill” string type information.
  - Libc string functions “gen” string type information.
  - Strings at entry to CFG are constant strings or function parameters.

# Problem 1: No Types [example]

String variables:

- after the call: { eax }
- before the call: { ebx, ecx }

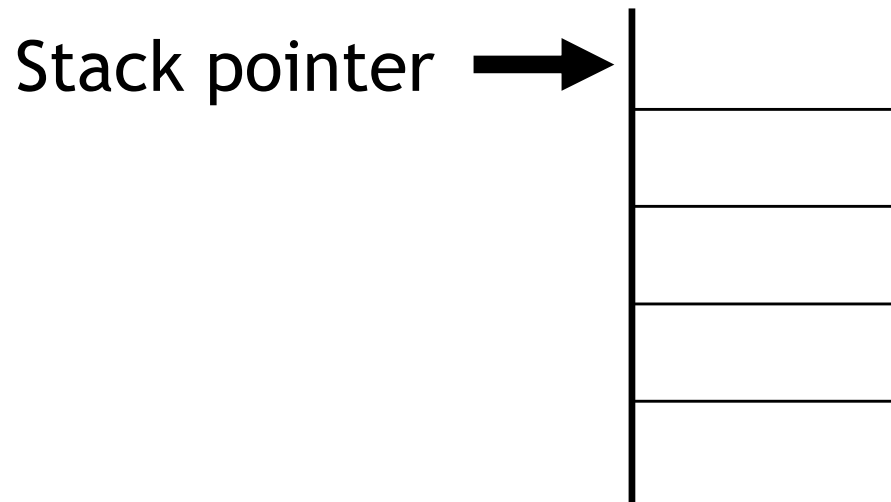
```
eax = _strcat( ebx, ecx );
```

# Problem 2: Function Parameters

- Function parameters are not explicit in x86 machine code.

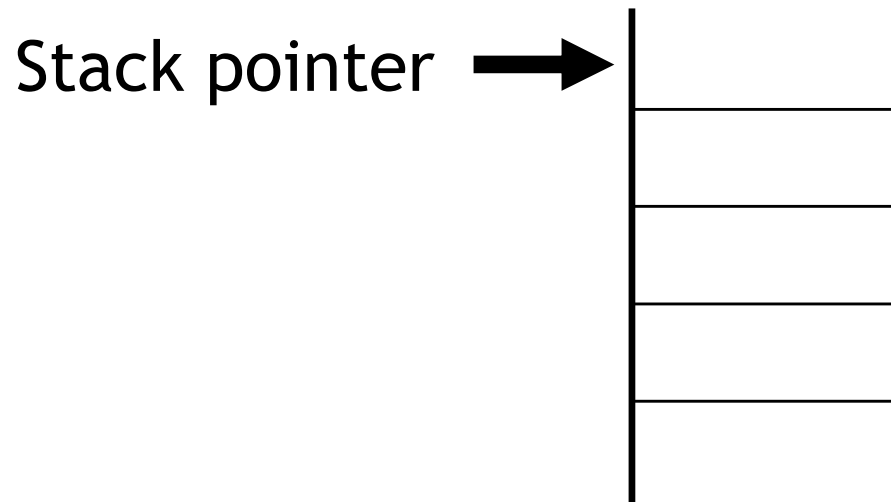
```
mov    ecx, [ebp+var1]
push  ecx
mov    ebx, [ebp+var2]
push  ebx
call  _strcat
add   esp, 8
```

# Problem 2: Fn. Params [example]



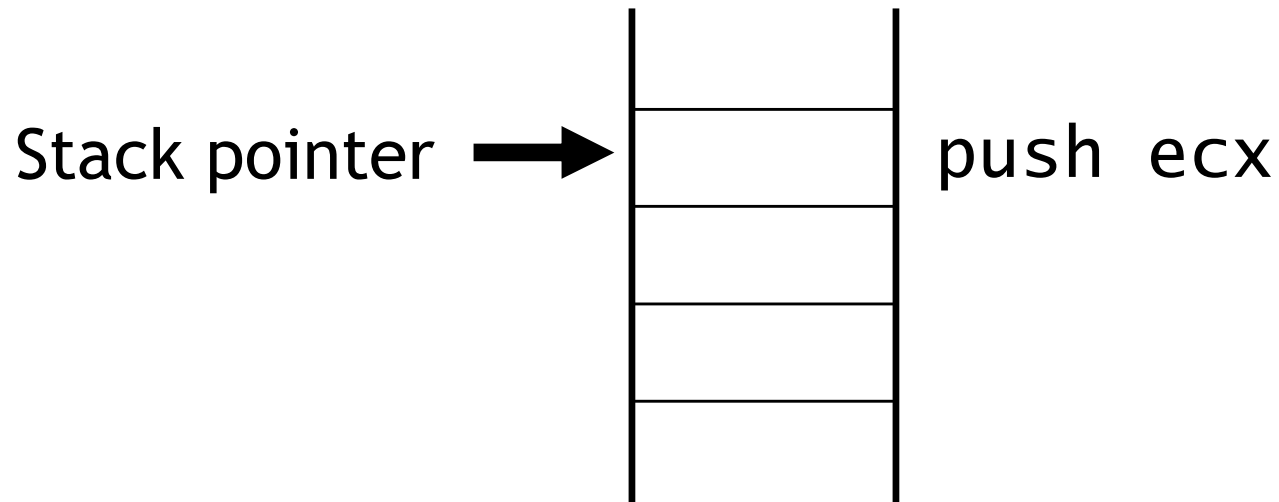
```
→ mov ecx, [ebp+var1]
   push ecx
   mov ebx, [ebp+var2]
   push ebx
   call _strcat
   add esp, 8
```

# Problem 2: Fn. Params [example]



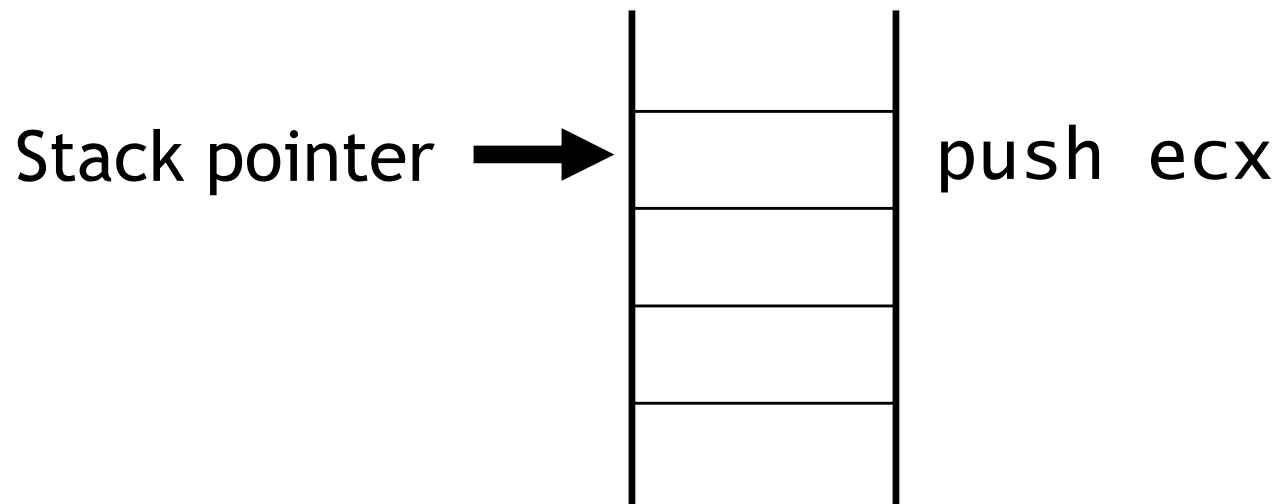
```
→ mov ecx, [ebp+var1]
   push ecx
   mov ebx, [ebp+var2]
   push ebx
   call _strcat
   add esp, 8
```

# Problem 2: Fn. Params [example]



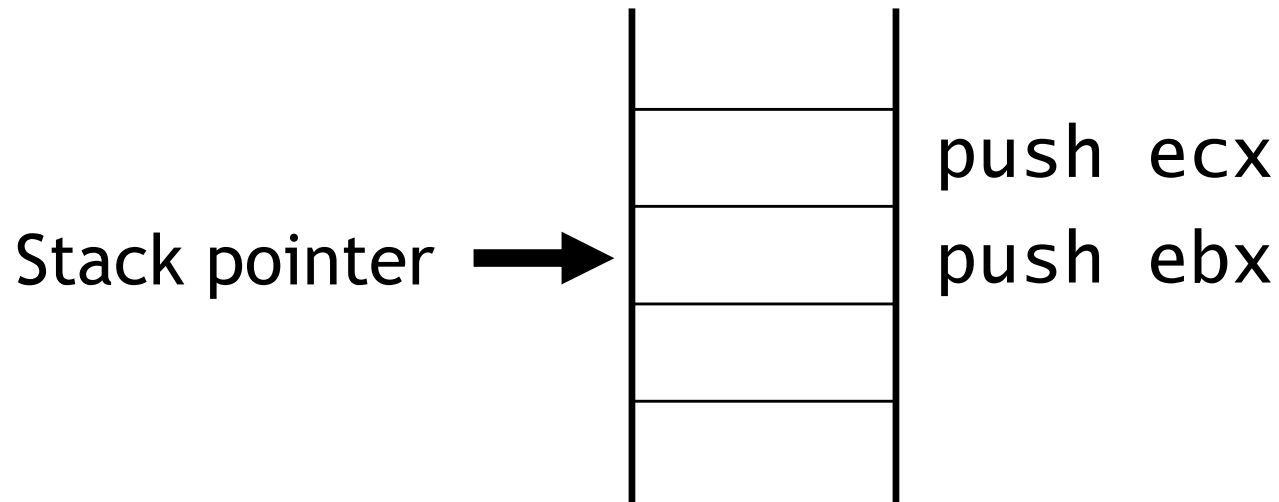
```
mov ecx, [ebp+var1]
push ecx
→ mov ebx, [ebp+var2]
push ebx
call _strcat
add esp, 8
```

# Problem 2: Fn. Params [example]



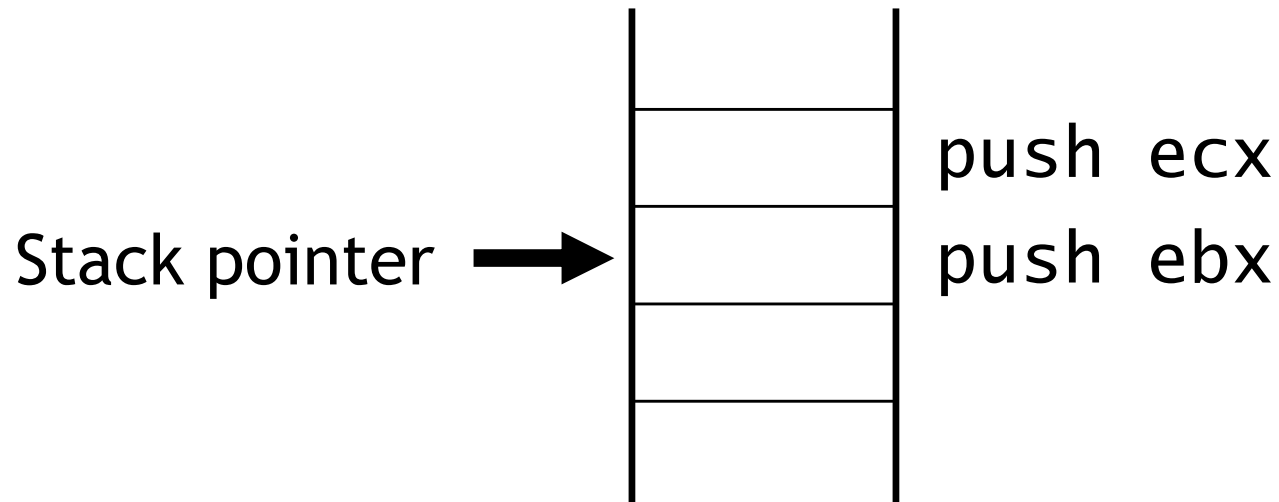
```
mov ecx, [ebp+var1]
push ecx
→ mov ebx, [ebp+var2]
push ebx
call _strcat
add esp, 8
```

# Problem 2: Fn. Params [example]



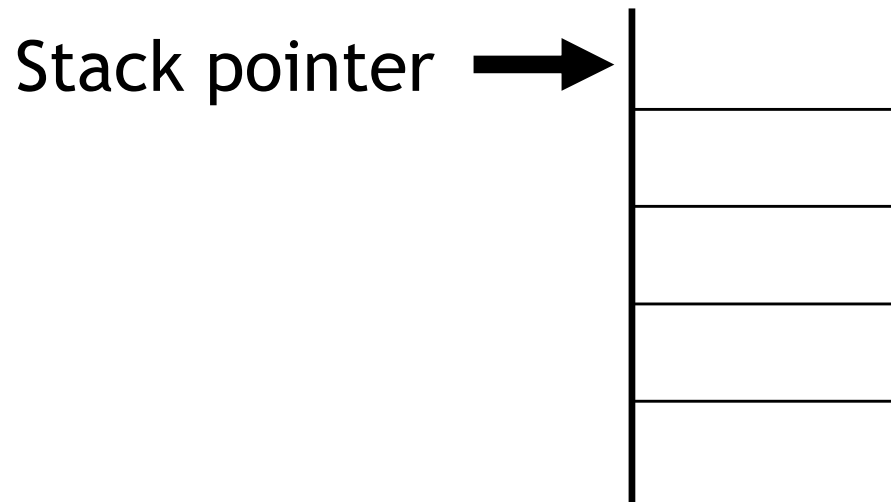
```
mov ecx, [ebp+var1]
push ecx
mov ebx, [ebp+var2]
push ebx
→ call _strcat
add esp, 8
```

# Problem 2: Fn. Params [example]



```
mov ecx, [ebp+var1]
push ecx
mov ebx, [ebp+var2]
push ebx
→ call _strcat
add esp, 8
```

# Problem 2: Fn. Params [example]

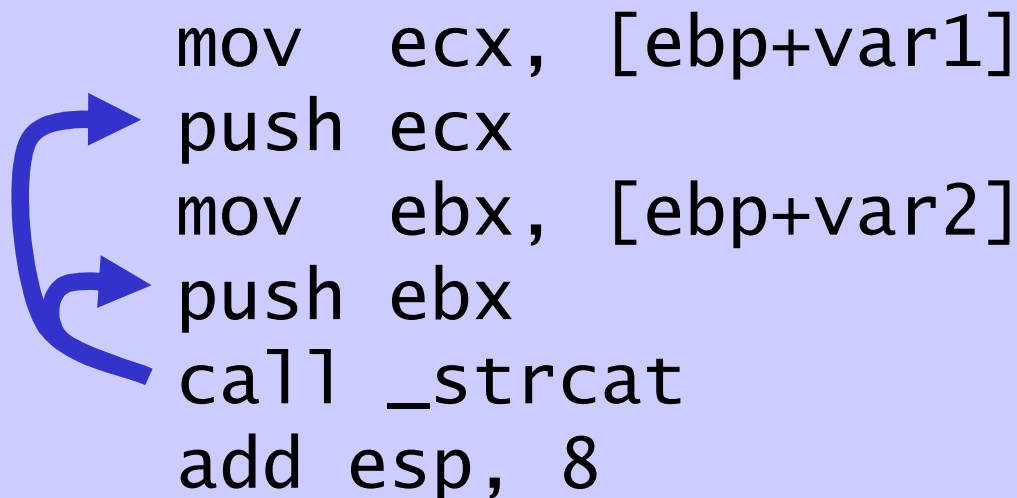


```
mov ecx, [ebp+var1]
push ecx
mov ebx, [ebp+var2]
push ebx
call _strcat
add esp, 8
```



# Problem 2: Function Parameters

- Solution: Perform forwards analysis modeling x86 instructions effects on the stack.



```
mov    ecx, [ebp+var1]
push  ecx
mov    ebx, [ebp+var2]
push  ebx
call  _strcat
add   esp, 8
```

# Problem 3: Unmodeled Functions

- String type information and stack model may be incorrect!

Assumption #2:

“\_cdecl” calling convention and well behaved functions

- Treat all function arguments and return values as strings.

# Problem 4: Java vs. x86 Semantics

- Java strings are immutable, x86 strings are not.

```
String y, x="x";  
y = x;  
y = y + "123";  
System.out.println(x);
```

=> "x"

# Problem 4: Java vs. x86 Semantics

- Java strings are immutable, x86 strings are not.

```
String y, x="x";  
y = x;  
y = y + "123";  
System.out.println(x);
```

=> "x"

```
char *y;  
char x[10] = "x";  
y = x;  
y = strcat(y, "123");  
printf(x);
```

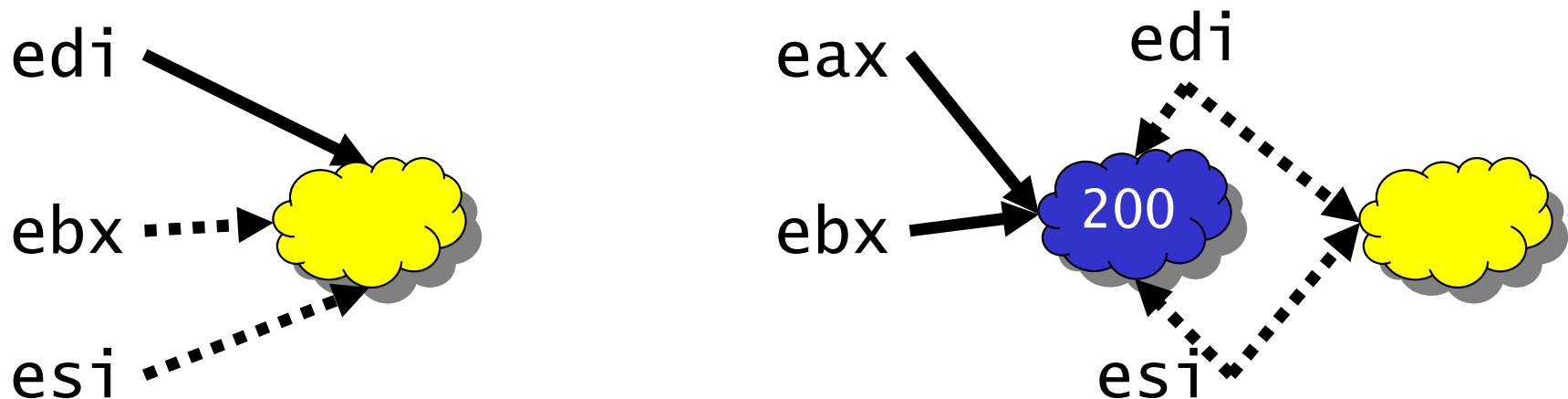
=> "x123"

# Problem 4: Java vs. x86 Semantics

- Solution: May-Must alias analysis.

```
0x200: _strcat( ebx, ecx )
```

“**May** alias” relations:

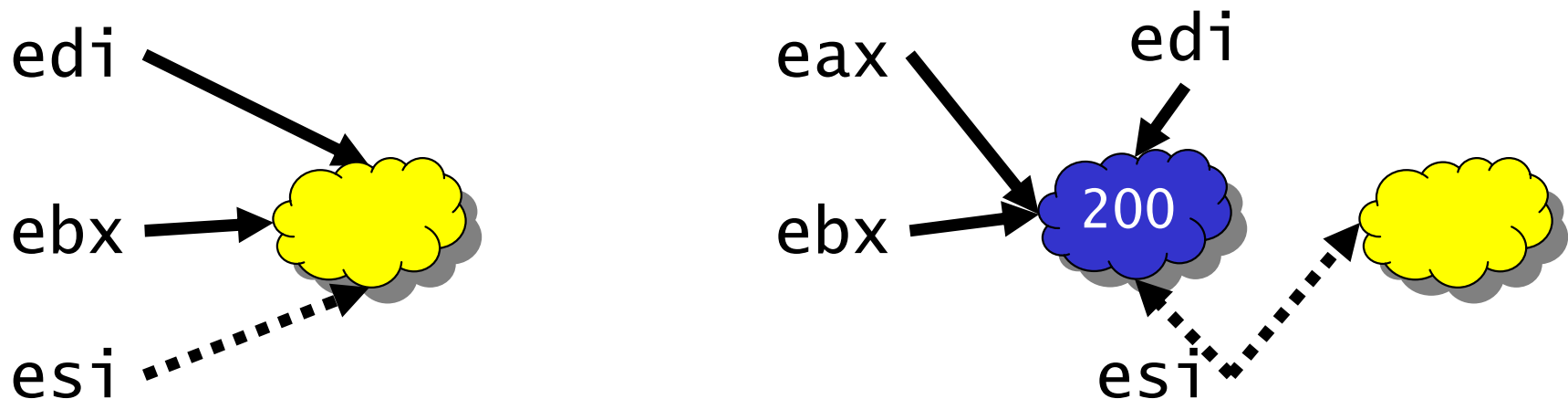


# Problem 4: Java vs. x86 Semantics

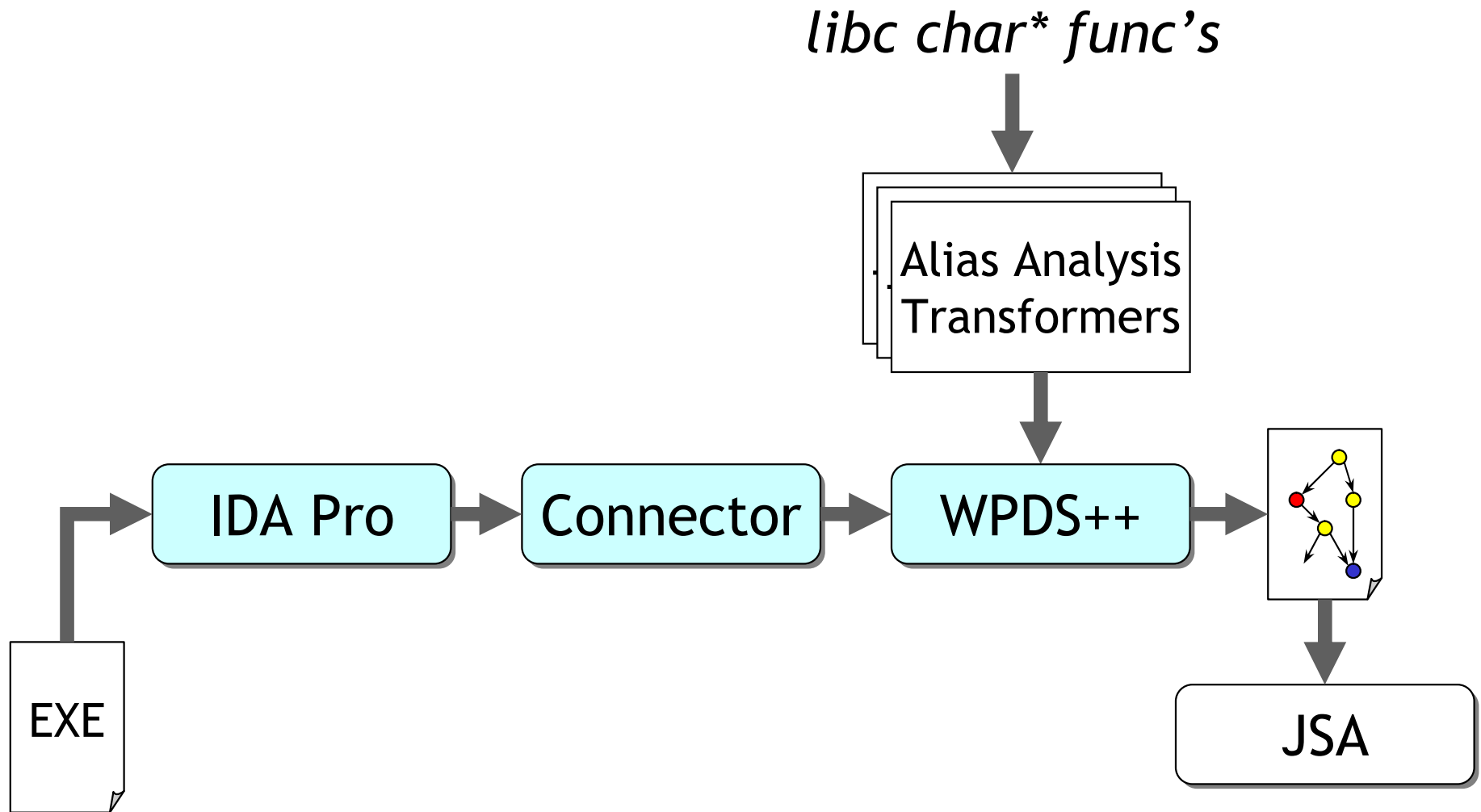
- Solution: May-Must alias analysis.

```
0x200: _strcat( ebx, ecx )
```

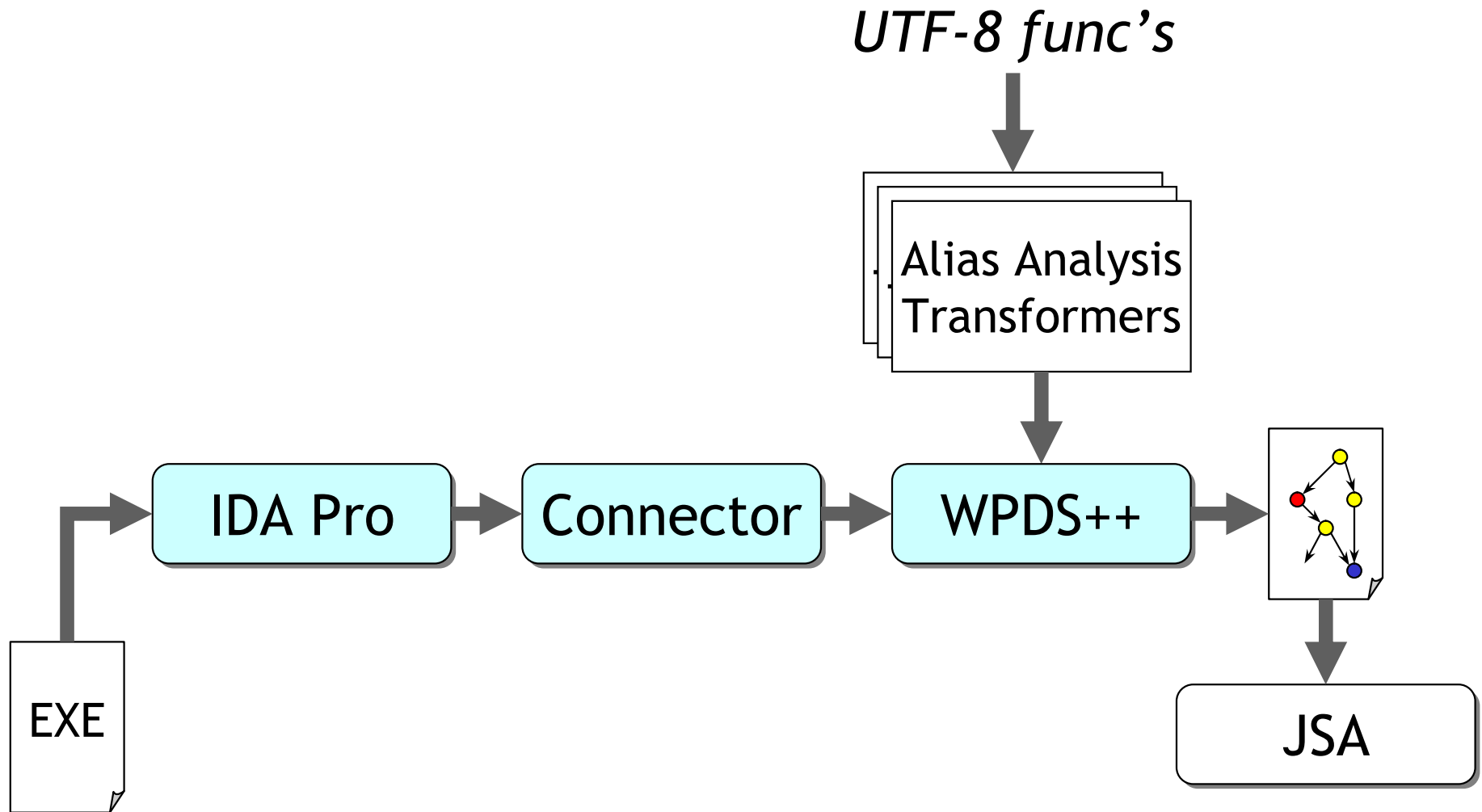
“**Must** alias” relations:



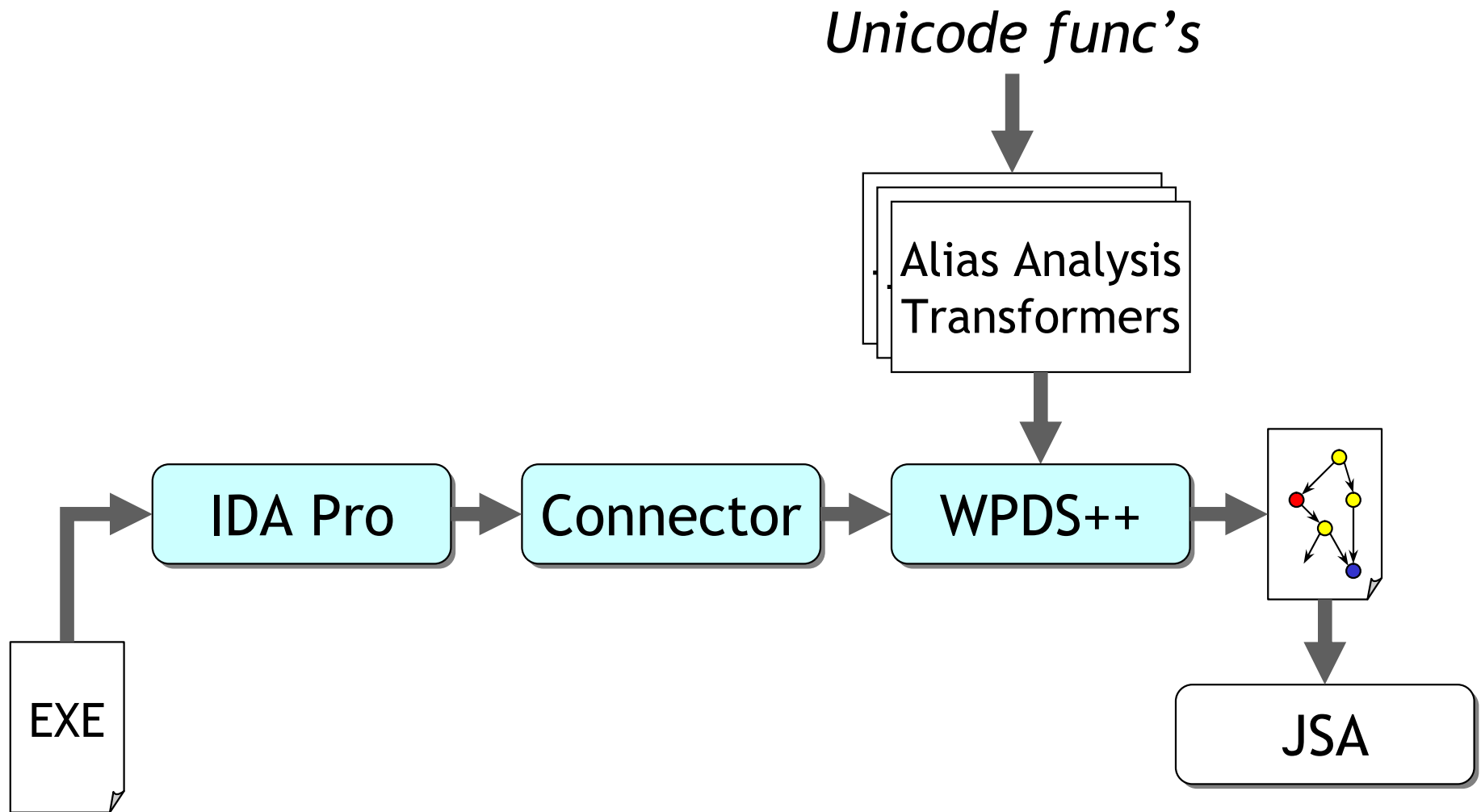
# x86sa Architecture



# x86sa Architecture



# x86sa Architecture



# Intraprocedural Analysis Summary

1. Recover callsite arguments.  
(stack-operation modeling)
  2. Infer string types.  
(backward type analysis)
  3. Discover aliases.  
(may-, must-alias forward analysis)
- ✓ Generate the String Flow Graph for the Control Flow Graph.

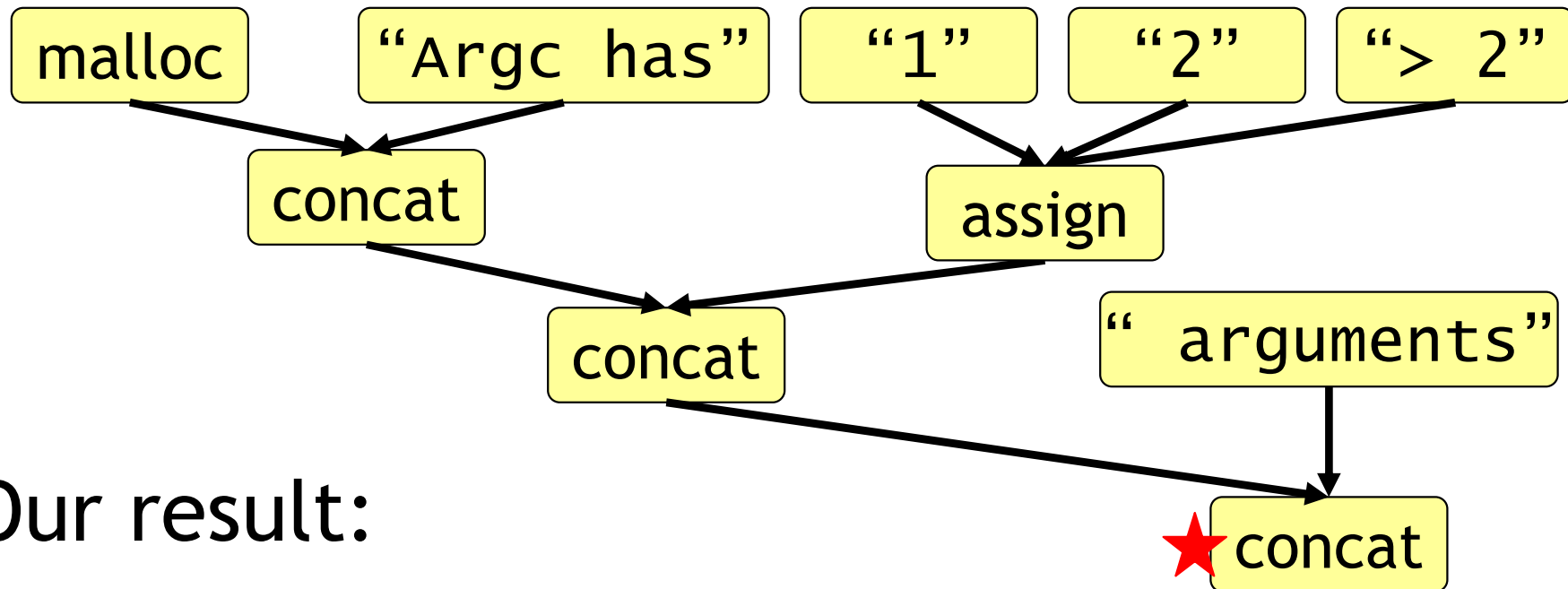
# Outline

- String analysis for Java.
- String analysis for x86.
- Evaluation.
- Applications & future work.

# Example 1: simple

```
char * s1 = "Argc has ";
char * s2;
char * s3 = " arguments";
char * s4;
switch( argc ) {
    case 1:    s2 = "1"    ; break;
    case 2:    s2 = "2"    ; break;
    default:   s2 = "> 2"; break;
}
s4 = malloc( strlen(s1)+strlen(s2)+strlen(s3)+1 );
s4[0] = 0;
strcat( strcat( strcat( s4, s1 ), s2), s3 );
★ printf( "%s\n", s4 );
```

# Example 1: String Flow Graph



Our result:

"Argc has 1 arguments"

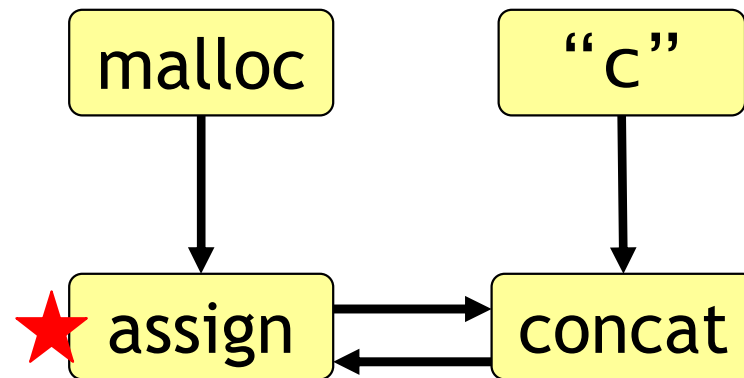
"Argc has 2 arguments"

"Argc has > 2 arguments"

# Example 2: `cstar`

```
char * c = "c";  
char * s4 = malloc(101);  
for( int i=0; i < 100 ; i++ ) {  
    strcat( s4, c );  
}  
★ printf( "%s\n", s4 );
```

# Example 2: String Flow Graph



Our result:  $c^*$

Correct answer:  $c^{100}$

# Example 3: Lion Worm

- Code and String Flow Graph omitted.
- x86sa analysis results:  

```
"/sbin/ifconfig -a|/bin/mail  
angelz1578@usa.net"
```

# Future Work:

## Interprocedural Analysis

1. Inline everything and apply intra-procedural analysis.
2. “Hook” intraprocedural String Flow Graphs into a “Super String Flow Graph”.
3. Polyvariant analysis over function summaries for String Flow Graphs.

# Future Work:

## Relax Assumptions

- Relax the assumptions:
  - Strings can be manipulated in many ways.
  - Calling conventions can vary in a program.
- **Value Set Analysis** looks promising:
  - Identifies “variables” based on usage patterns.

# Future Work:

## More Applications

- Malicious code analysis
- Analysis of dynamic code generators:
  - Packed programs
  - Shell code generators

# String Analysis for Binaries

Mihai Christodorescu

Nicholas Kidd

Wen-Han Goh

University of Wisconsin, Madison

{ mihai, [kidd](mailto:kidd@cs.wisc.edu), wen-han }@[cs.wisc.edu](mailto:cs.wisc.edu)