Efficient Reflection String Analysis via Graph Coloring

Neville Grech
George Kastrinis
Yannis Smaragdakis

Strings important for static analysis?

```
s1 = “script error in file {0} : {1}”

s2 = “count”

s3 = “Usage: {0} [options] [arguments...]\n\nwhere...”

s4 = “Manager”
```

Enter reflection

```
s1 = “script error in file {0} : {1}”

s2 = “count”

s3 = “Usage: {0} [options] [arguments...]\n\nwhere...”

s4 = “Manager”

Class c = Class.forName(s4)

Method m = c.getMethod(s2 + “Sales”)

m.invoke(...)
```

Reflection - The backbone of dynamic features

- e.g. Dynamic Proxy Pattern in Java (~ 21% of open source programs)
- ignoring reflection ⇒ top causes of unsoundness
- highly controlled through string values (member selectors)

Naive string analysis is expensive

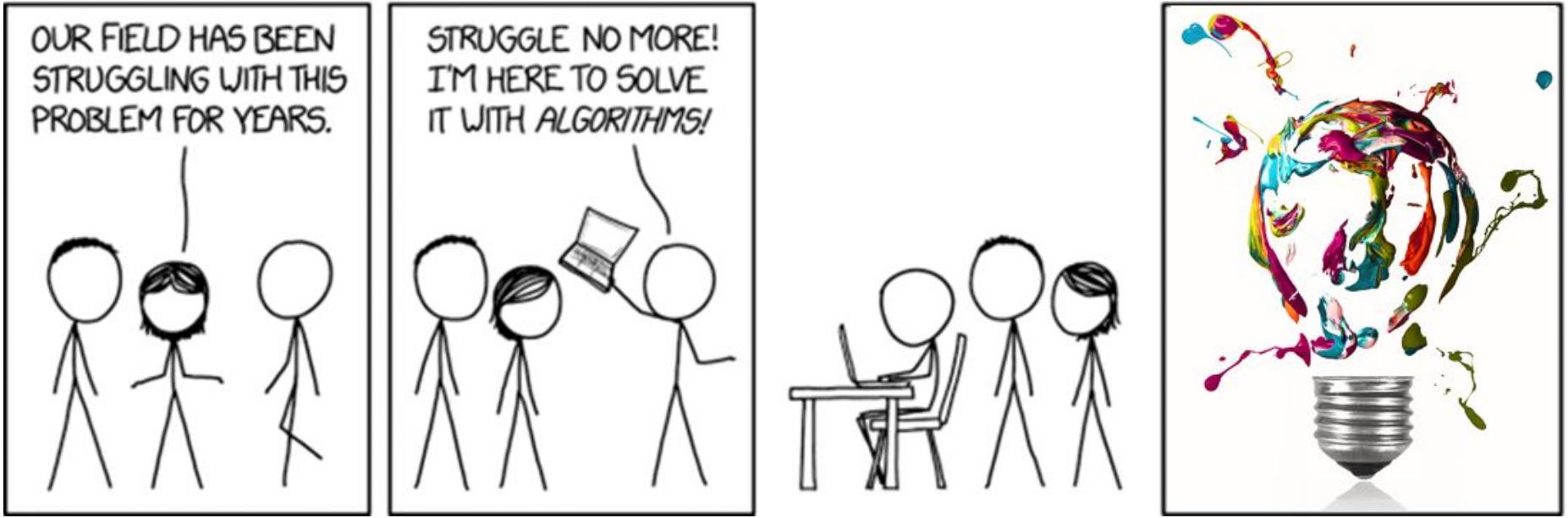
- Doop & DaCapo-Bach avrora (context-insensitive):

  2.9M Strings vs 2M regular objects in Var-Points-To

- IBM WALA & DaCapo-2006 antlr (0-1-CFA):

  6.7M vs 1.7M

  dominated by string values


OUR FIELD HAS BEEN STRUGGLING WITH THIS PROBLEM FOR YEARS.
STRUGGLE NO MORE! I'M HERE TO SOLVE IT WITH *ALGORITHMS!*


compress String constants
Retain member selection ability


SNEAK PEEK

DOOP

SOLAR

DOOP

SOLAR

SIZE REDUCTION FOR POINTS-TO SETS (WITH STRING VALUES)

OVER VERY **AGGRESSIVE** STRING INTERNING TECHNIQUES

~2x

DOOP

SIZE REDUCTION FOR POINTS-TO SETS (WITH STRING VALUES)

~2x

OVER VERY **AGGRESSIVE** STRING INTERNING TECHNIQUES

SIZE REDUCTION FOR COMPUTED SETS

~1.5x

DOOP

SIZE REDUCTION FOR POINTS-TO SETS (WITH STRING VALUES)

~2X

OVER VERY **AGGRESSIVE** STRING INTERNING TECHNIQUES

SIZE REDUCTION FOR COMPUTED SETS

~1.5X

SPEEDUP

~20%

DOOP

SOLAR

SIZE REDUCTION FOR POINTS-TO SETS (WITH STRING VALUES)

~2X

OVER VERY **AGGRESSIVE** STRING INTERNING TECHNIQUES

SIZE REDUCTION FOR COMPUTED SETS

~1.5X

SPEEDUP

~20%

TRANSPARENT APPROACH - NO PITFALLS!!

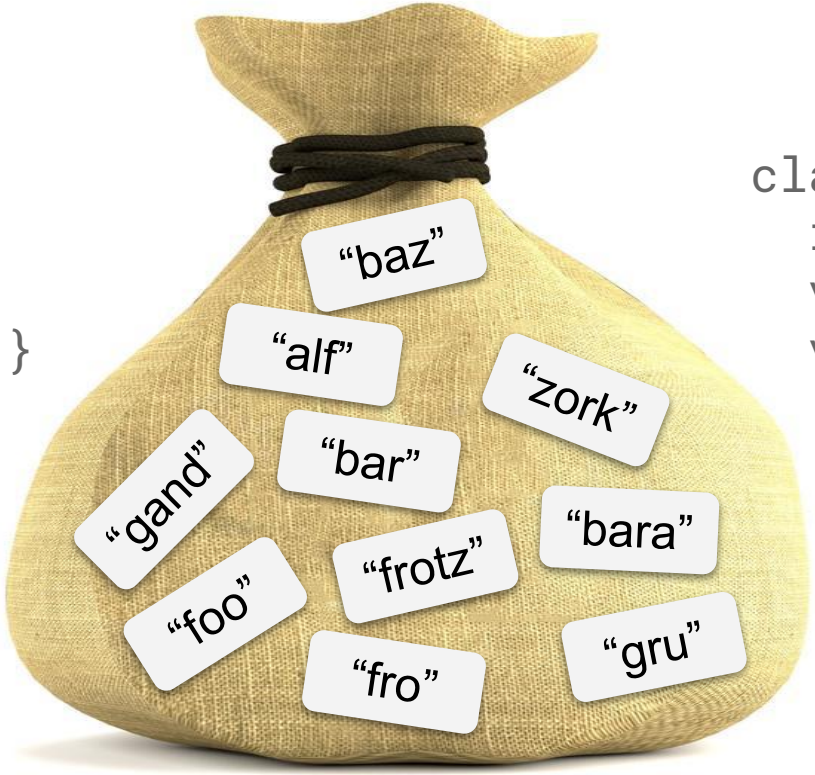
The idea - Color a conflict graph

- string constants as nodes
- edge iff two nodes **match*** distinct members in same class
- fast graph **coloring** (?)
- nodes with the same color can be **merged**

```
class B {
  int frotz;
  int grue;
  String zork() {..}}
```

```
class A {
  int foo;
  void bar() {..}
  void baz() {..}}
```

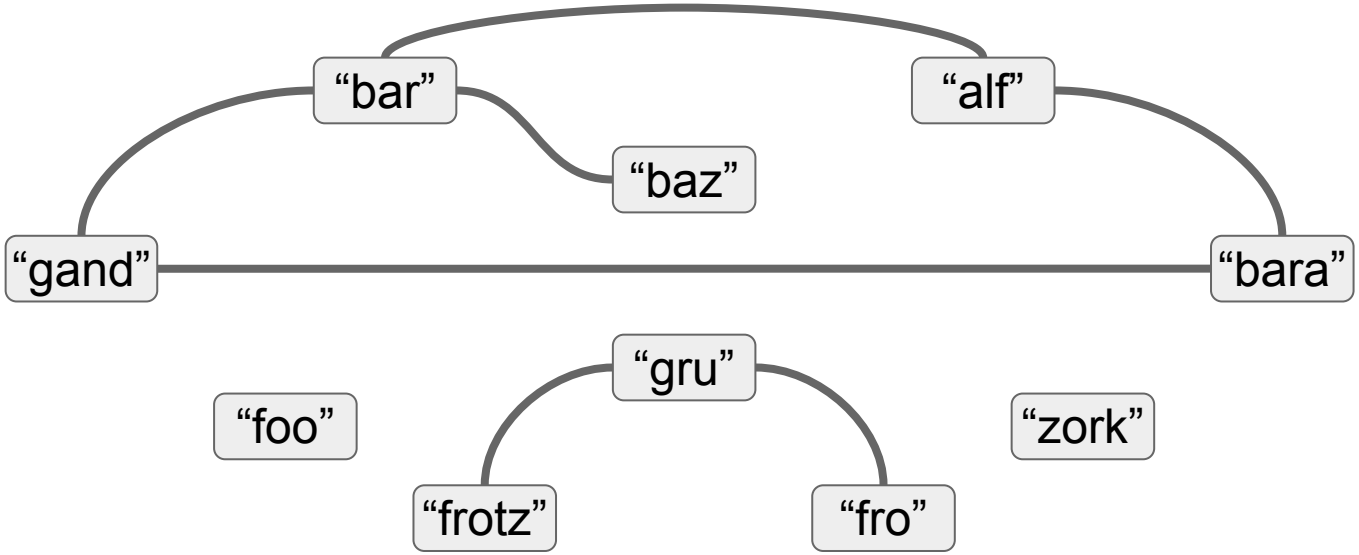
```
class C {
  int frodo;
  void gandalf() {..}
  void barahir() {..}}
```



```
class A {
  int foo;
  void bar() {..}
  void baz() {..}}
```

```
class B {
  int frotz;
  int grue;
  String zork() {..}}
```

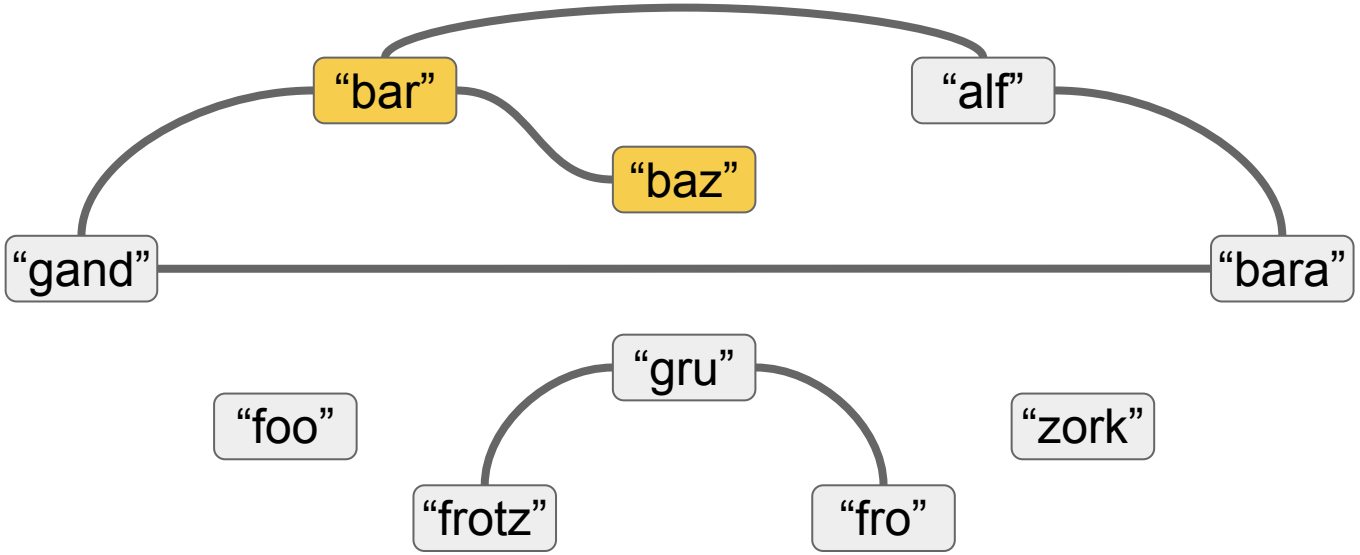
```
class C {
  int frodo;
  void gandalf() {..}
  void barahir() {..}}
```



```
class A {
  int foo;
  void bar() {..}
  void baz() {..}}
```

```
class B {
  int frotz;
  int grue;
  String zork() {..}}
```

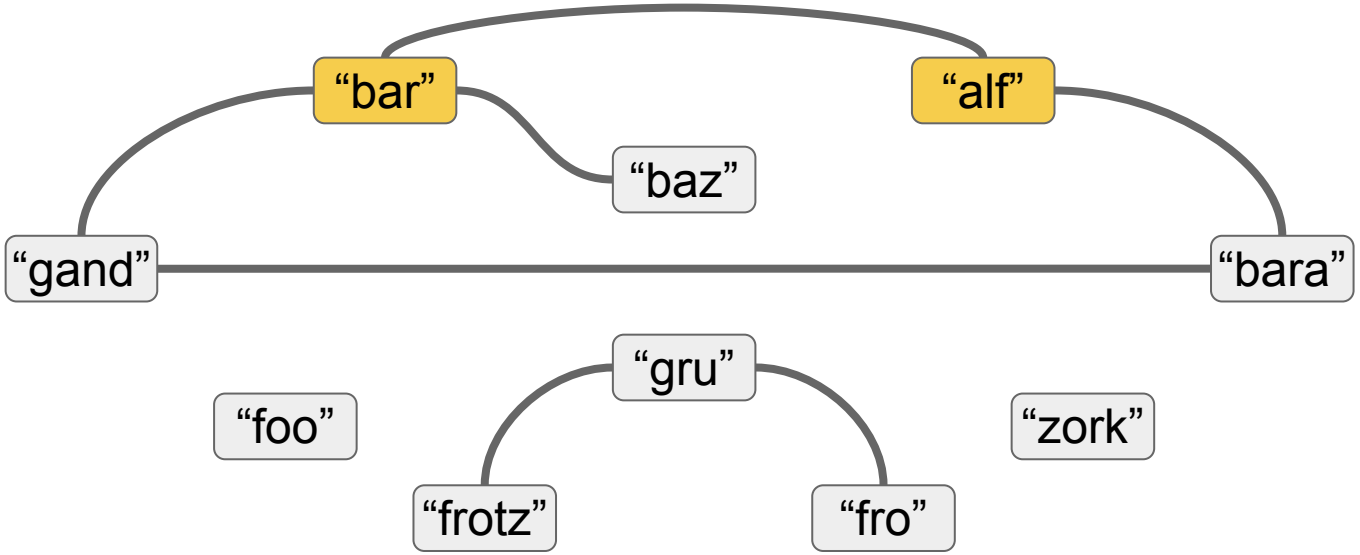
```
class C {
  int frodo;
  void gandalf() {..}
  void barahir() {..}}
```



```
class A {
  int foo;
  void bar() {..}
  void baz() {..}}
```

```
class B {
  int frotz;
  int grue;
  String zork() {..}}
```

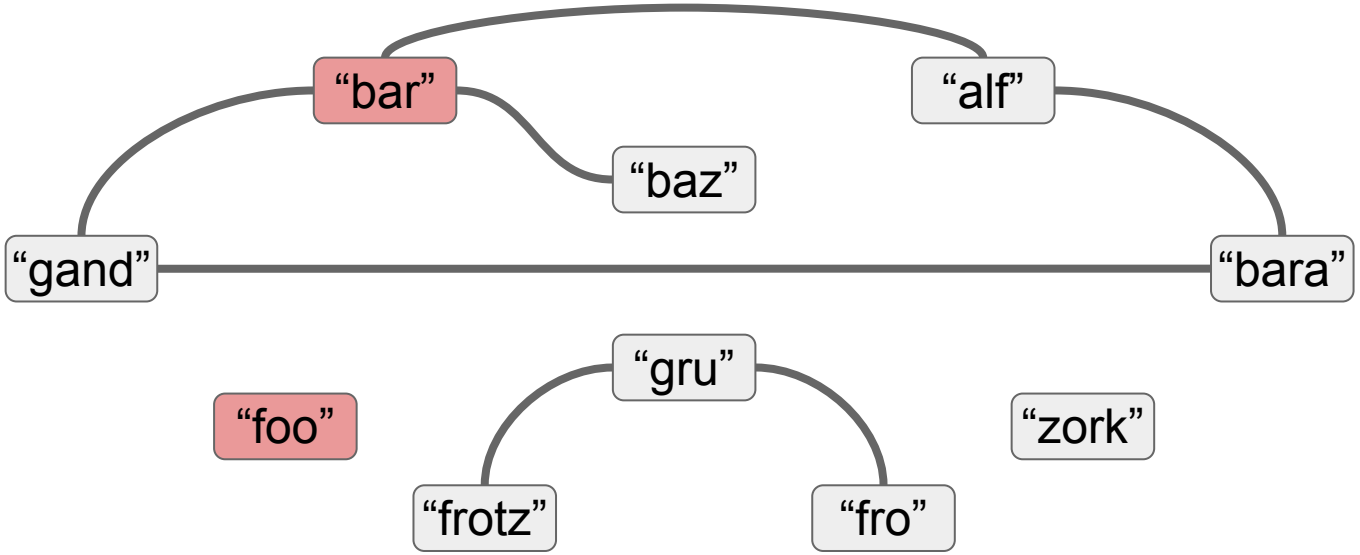
```
class C {
  int frodo;
  void gandalf() {..}
  void barahir() {..}}
```



```
class A {
  int foo;
  void bar() {..}
  void baz() {..}}
```

```
class B {
  int frotz;
  int grue;
  String zork() {..}}
```

```
class C {
  int frodo;
  void gandalf() {..}
  void barahir() {..}}
```



Suboptimal is good enough

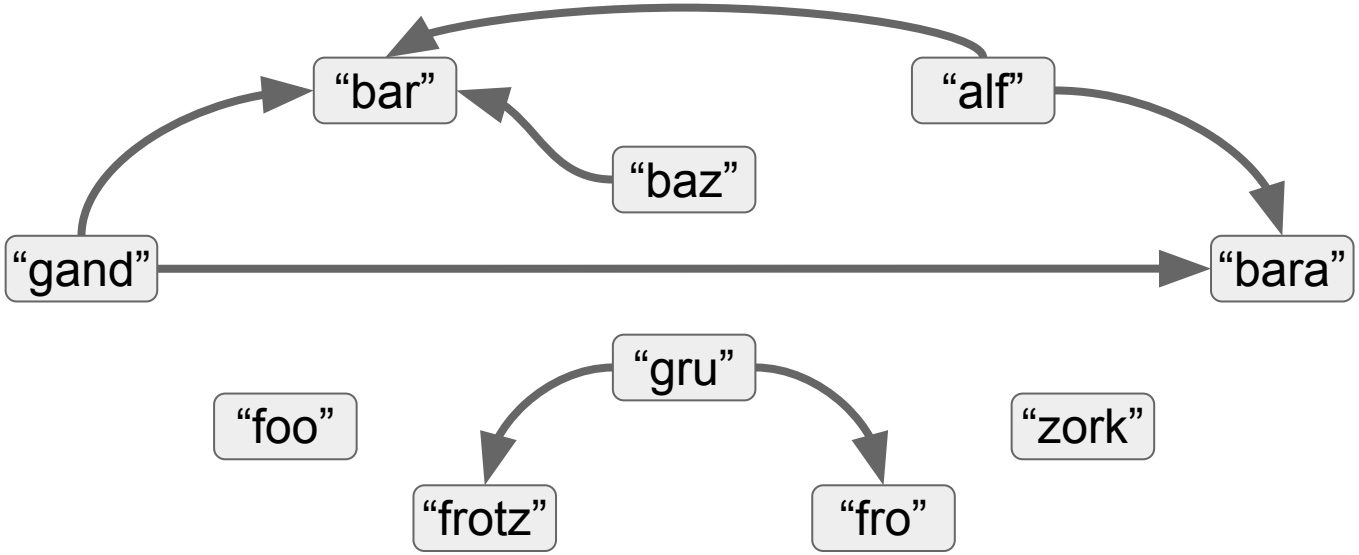
- MINIMUM #colors required already too large
- several THOUSANDS (strings) ⇒ few HUNDREDS (colors) already beneficial
- benefit NOT PROPORTIONAL to the reduction

NEAR-LINEAR-time greedy algorithm

```
class A {
  int foo;
  void bar() {..}
  void baz() {..}}
```

```
class B {
  int frotz;
  int grue;
  String zork() {..}}
```

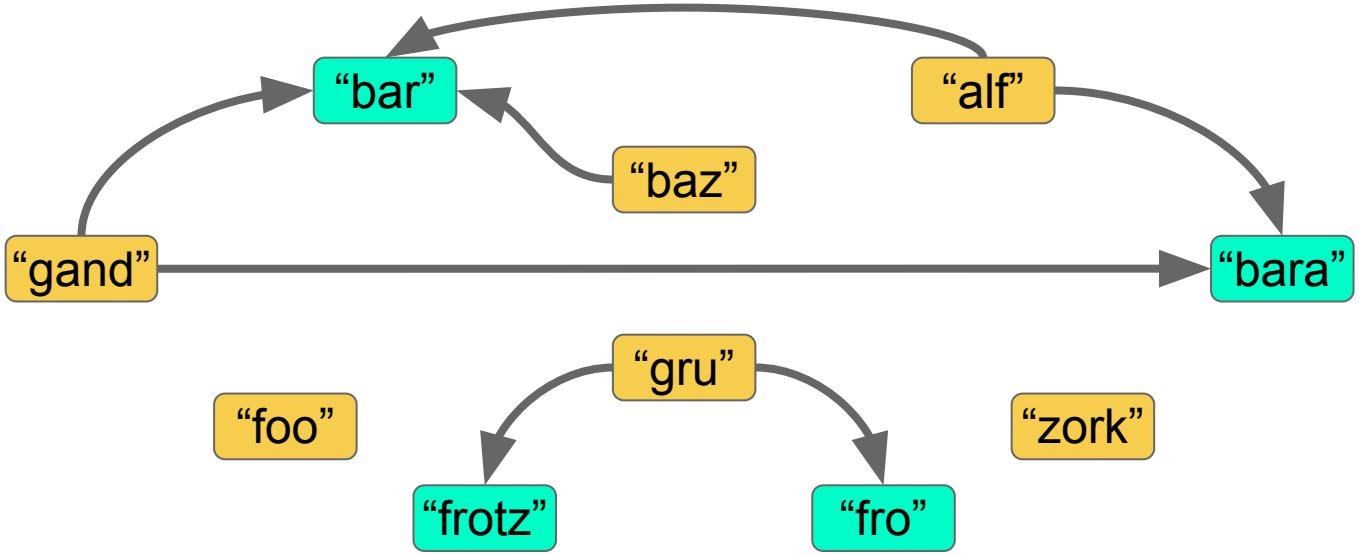
```
class C {
  int frodo;
  void gandalf() {..}
  void barahir() {..}}
```



```
class A {
  int foo;
  void bar() {..}
  void baz() {..}}
```

```
class B {
  int frotz;
  int grue;
  String zork() {..}}
```

```
class C {
  int frodo;
  void gandalf() {..}
  void barahir() {..}}
```




“zork”
“gand”
“alf”
“baz”
“foo”
“gru”
“bar”
“fro”
“bara”
“frotz”


UNDER THE HOOD

Before

```
String a = “zork”;

…

Class cls = unknown() ? A.getClass() : B.getClass();

…

Method m = cls.getMethod(a);
```

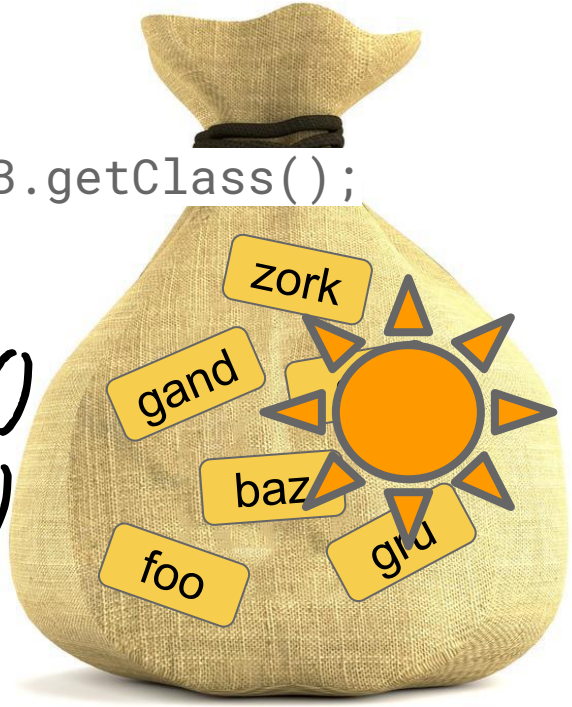
B:zork()

AFTER - UNWANTED IMPRECISION

```
String a = ☀;

…

Class cls = unknown() ? A.getClass() : B.getClass();

…

Method m = cls.getMethod(a);
```

B:zork()

A:baz()



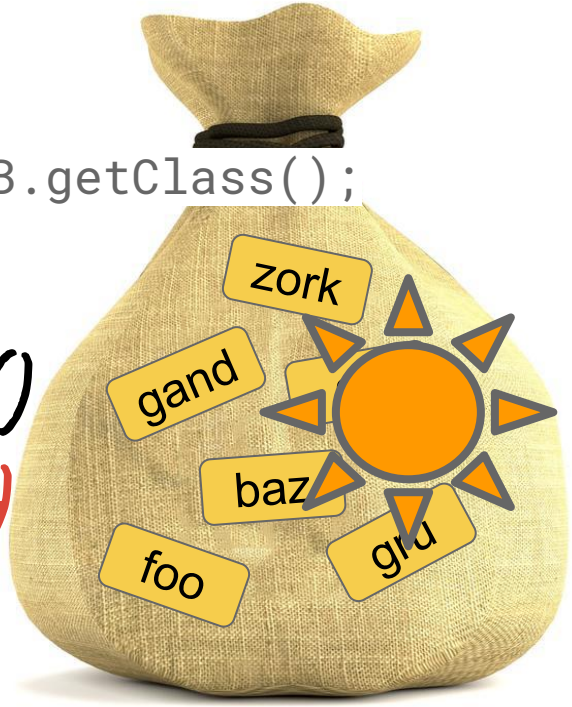
BACKWARD ANALYSIS

```
String a = ☀;

…

Class cls = unknown() ? A.getClass() : B.getClass();

…

Method m = cls.getMethod(a);

String s = (String) m.invoke();
```

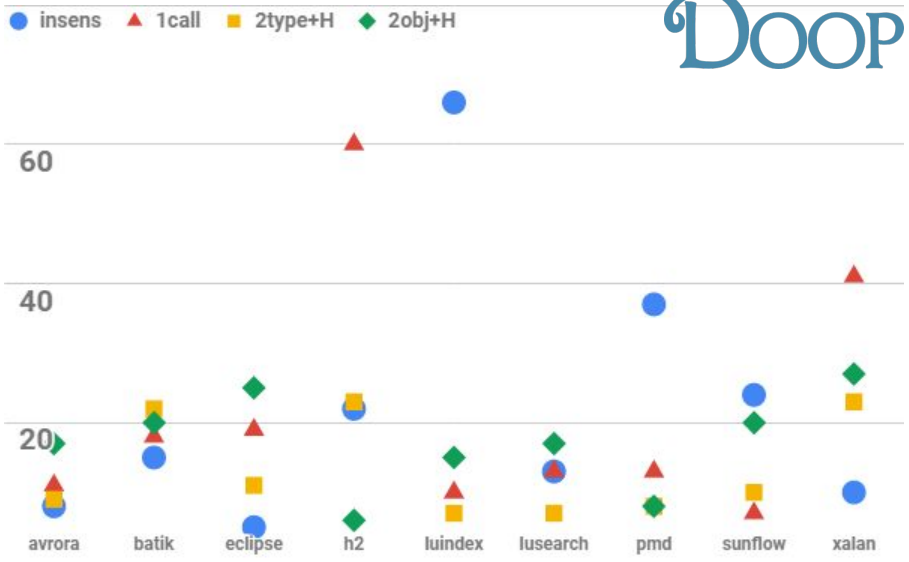
void A:baz()

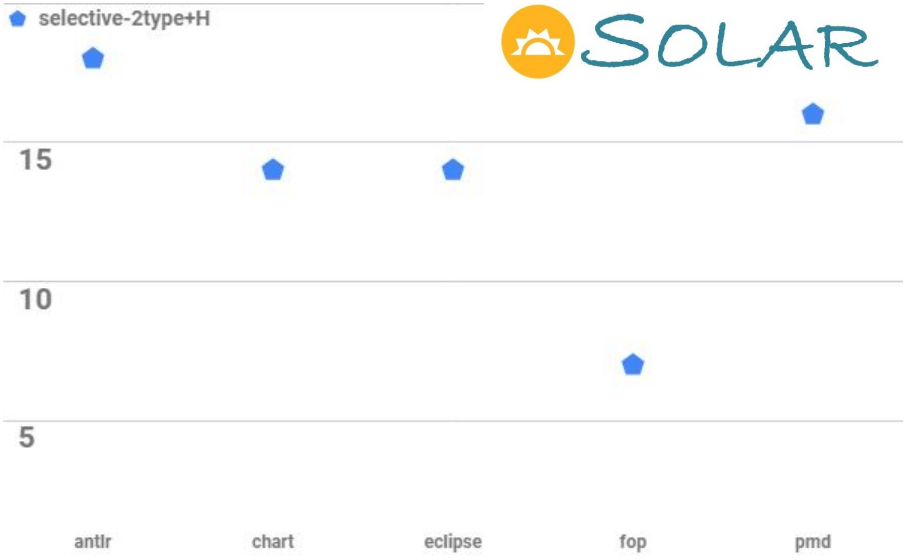
B:zork()

~~A:baz()~~




Results
Just Ahead

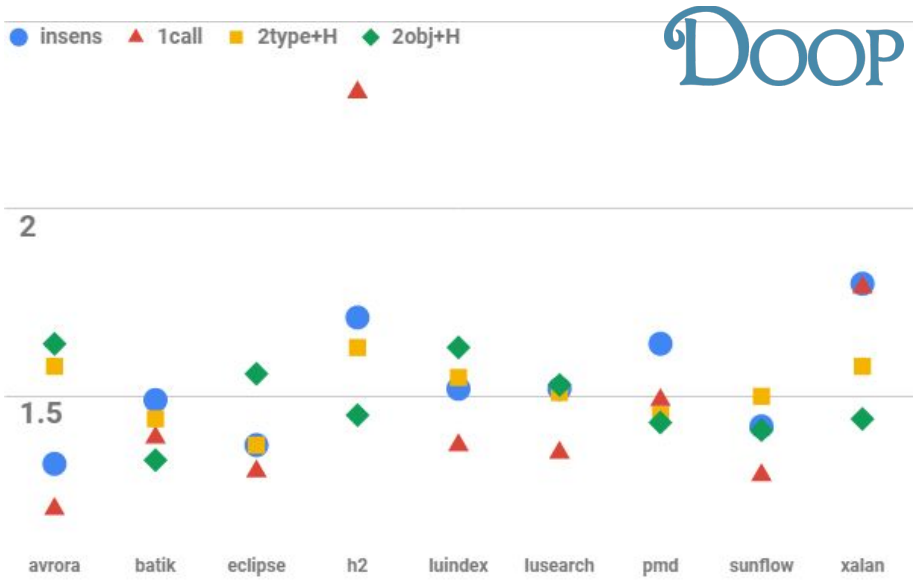


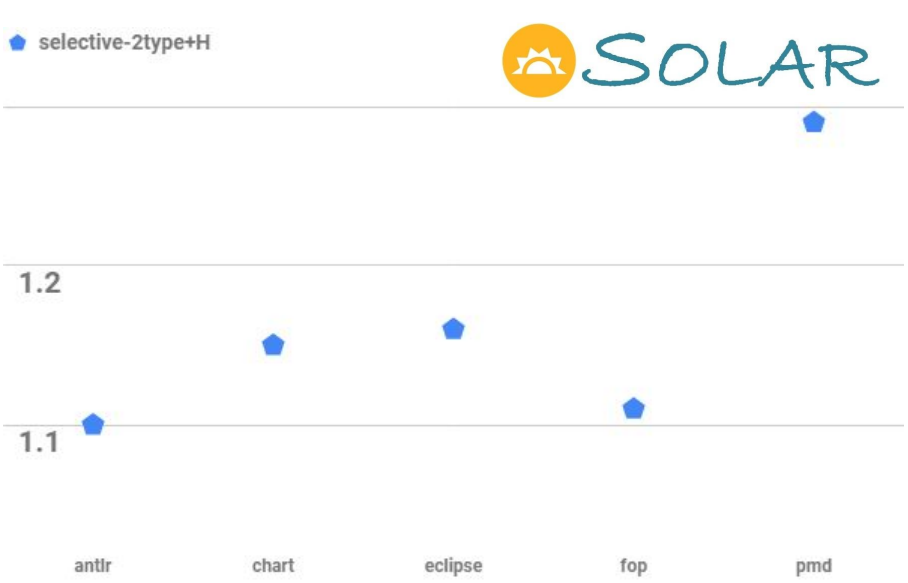


~20%

~14%

Analysis speedup

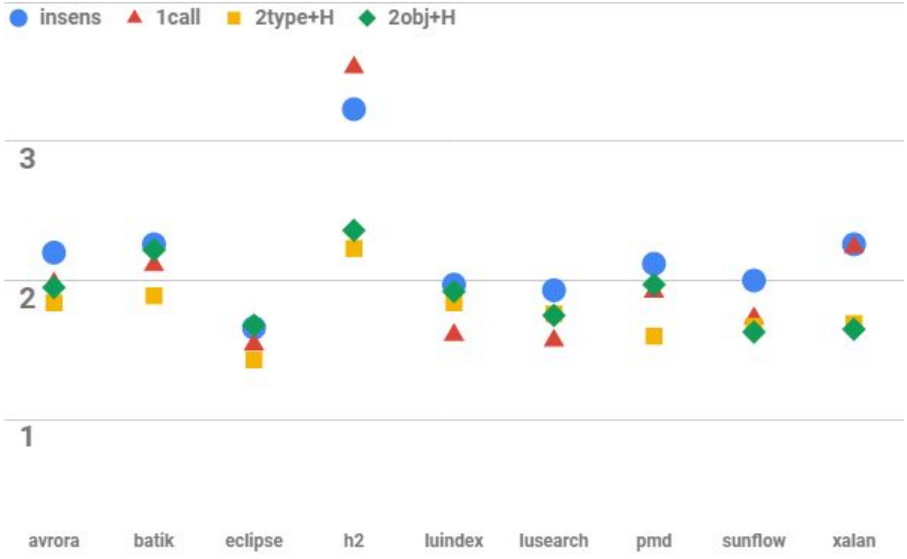




~1.5x

~1.16x

Memory reduction (Var-Points-To)



~1.97x

String Var-Points-To reduction

PRECISION & SOUNDNESS

$\varepsilon < 0.2\%$

IN MOST CASES ZERO

EFFECTIVENESS

~1SEC

COMPRESSION RATIO

~6.5X

- string values **important** in analyzing reflection

conclusion

- string values **important** in analyzing reflection
- they would **dominate** a naive analysis

conclusion

CONCLUSION

- STRING VALUES IMPORTANT IN ANALYZING REFLECTION
- THEY WOULD DOMINATE A NAIVE ANALYSIS
- COMPRESS WHILE RETAINING MEMBER SELECTION ABILITY (WITH NO PRACTICAL DRAWBACKS)


That's all Folks!