# 缺陷查找和修复技术

熊英飞

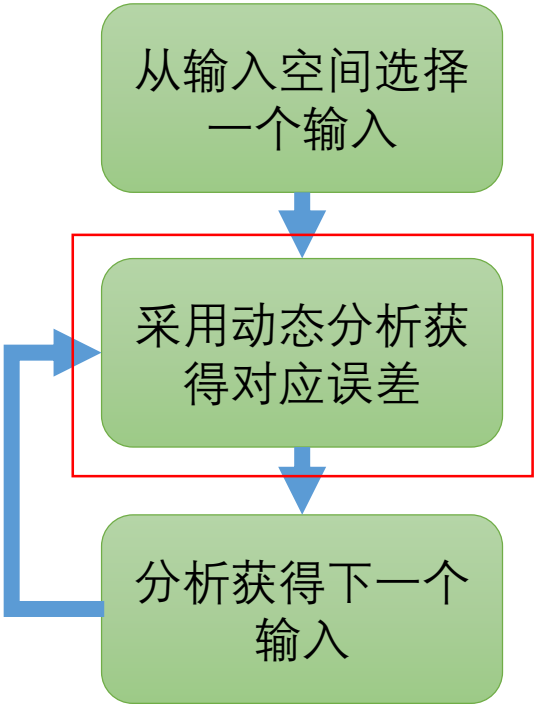
北京大学软件工程研究所

# 报告人介绍 – 熊英飞

- 2000~2004，电子科技大学本科
- 2004~2006，北京大学研究生
  - 导师：梅宏、杨芙清
- 2006~2009，日本东京大学博士
  - 导师：胡振江、武市正人
- 2009~2011，加拿大滑铁卢大学博士后
  - 导师：Krzysztof Czarnecki
- 2012~2018，北京大学预聘助理教授
- 2018~，北京大学长聘副教授
- 主要研究：自动/半自动发现和修复软件中的缺陷

# 查找缺陷：浮点误差测试技术

误差常常导致灾难性后果




**全自动查找程序中的浮点误差**
**发现GSL科学计算库中的多处潜在误差问题**
**发现最大误差数10倍于随机，3倍于遗传算法**

# 如何知道单次执行的误差?

Code:

```
float x = 0, a = 0.1;
for(int i = 0; i < 10000; i++) {
    x = x + a;
}
printf("%.6f", x);
```

→ $x_{orignal} = 999.902893$

↓ Raise precision

Code:

```
double x = 0, a = 0.1;
for(int i = 0; i < 10000; i++) {
    x = x + a;
}
printf("%.6lf", x);
```

→ $x_{high} = 1000.000000$

# 该方法真的可靠吗?

A code piece simplified from `exp` function in the GNU C library:

```
1:  double x = 3.7;
2:  double n = 6755399441055744.0;
3:  double y = (x + n) - n;
```

Answer from computer: `y = 4` 

Raise precision to long double:

Answer from computer: `y = 3.7002` 

# 该方法真的可靠吗?

A code piece simplified from `exp` function in the GNU C library:

```
1:  double x = 3.7;
2:  double n = 6755399441055744.0;
3:  double y = (x + n) - n;
```

The goal of this code is to round x to the nearest integer.

n is a “magic number” specially designed for double precision. ($n = 1.5 \times 2^{52}$)

# 该方法真的可靠吗?

A code piece simplified from `exp` function in the GNU C library:

```
1:  double x = 3.7;
2:  double n = 6755399441055744.0;
3:  double y = (x + n) - n;
```

A precision-specific operation

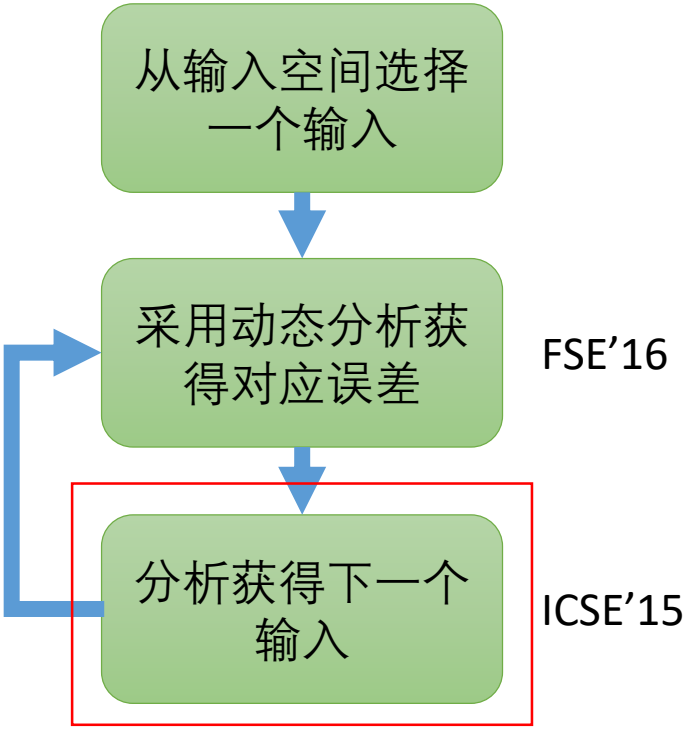
# Precision-Specific Semantics

```
double x = 3.7;
double n = 6755399441055744.0;
double y = (x + n) – n;
```

- Interpret as “rounding x to the nearest integer”
- Our Contribution
  - A heuristic to detect precision-specific operations.
  - A fixing approach to enable precision tuning under the presence of precision-specific operations.

# 查找缺陷：浮点误差测试技术

误差常常导致灾难性后果




**全自动查找程序中的浮点误差**
**发现GSL科学计算库中的多处潜在误差问题**
**发现最大误差数10倍于随机，3倍于遗传算法**

# 实证研究

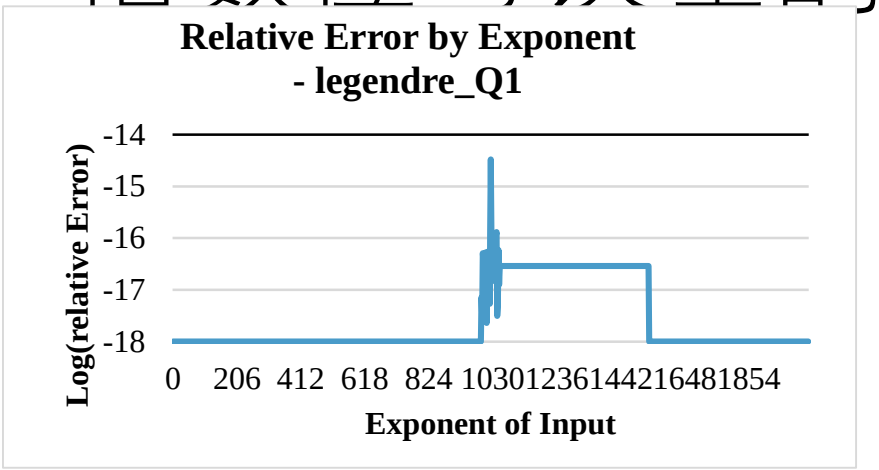
- 从GSL中选取4个函数，仅变化浮点数的尾数位或指数位，观察它们和内部误差之间的关系

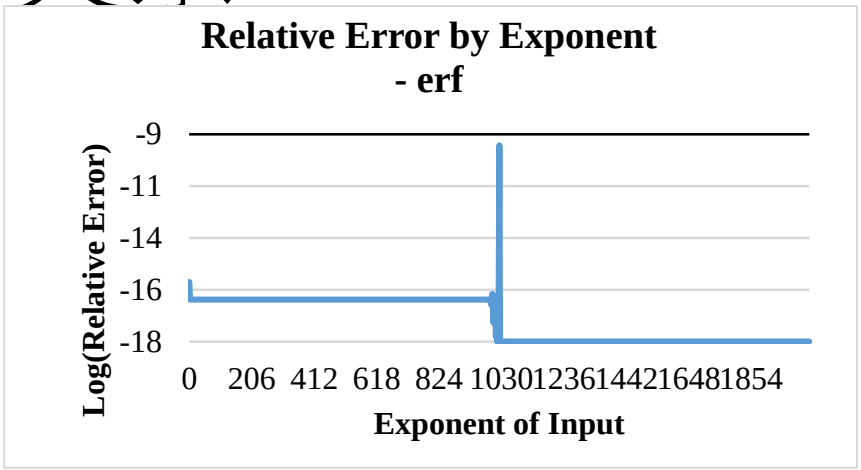
IEEE 754 Floating-Point Representation

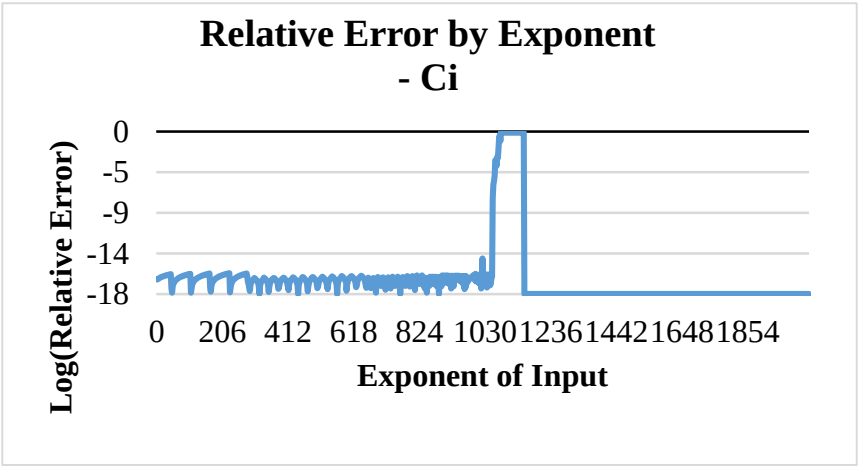
| | Sign | Exponent | Significand |
|---|---|---|---|
| **Single Precision** | 1 | 8 | 23 |
| **Double Precision** | 1 | 11 | 52 |

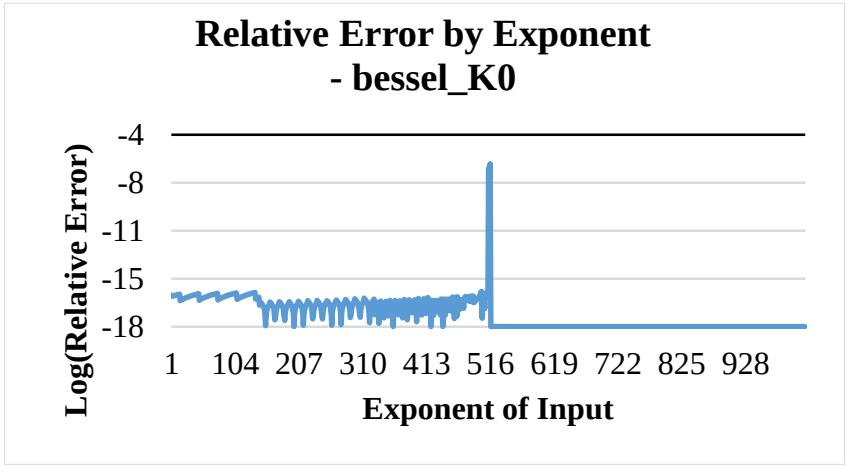
$$1.2345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10}_{\text{base}}^{\overbrace{-4}^{\text{exponent}}}$$

# 指数位与误差的关系


Relative Error by Exponent
- legendre_Q1
Log(relative Error)
-14
-15
-16
-17
-18
0 206 412 618 824 1030 1236 1442 1648 1854
Exponent of Input


Relative Error by Exponent
- erf
Log(Relative Error)
-9
-11
-14
-16
-18
0 206 412 618 824 1030 1236 1442 1648 1854
Exponent of Input


Relative Error by Exponent
- Ci
Log(Relative Error)
0
-5
-9
-14
-18
0 206 412 618 824 1030 1236 1442 1648 1854
Exponent of Input


Relative Error by Exponent
- bessel_K0
Log(Relative Error)
-4
-8
-11
-15
-18
1 104 207 310 413 516 619 722 825 928
Exponent of Input

# 主要发现 1

指数部分对误差的大小有重大影响。

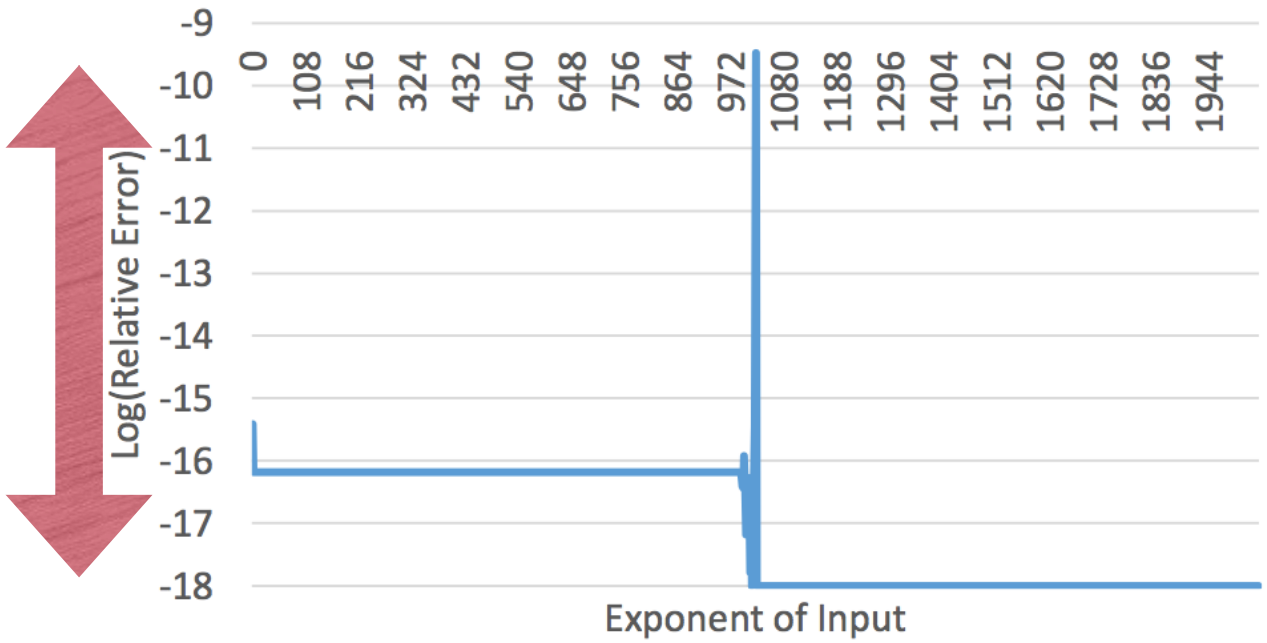



Fig. 1. erf at significand 0x34873b27b23c6

# 主要发现 2

能触发大误差的指数往往只存在于一个很小的范围。

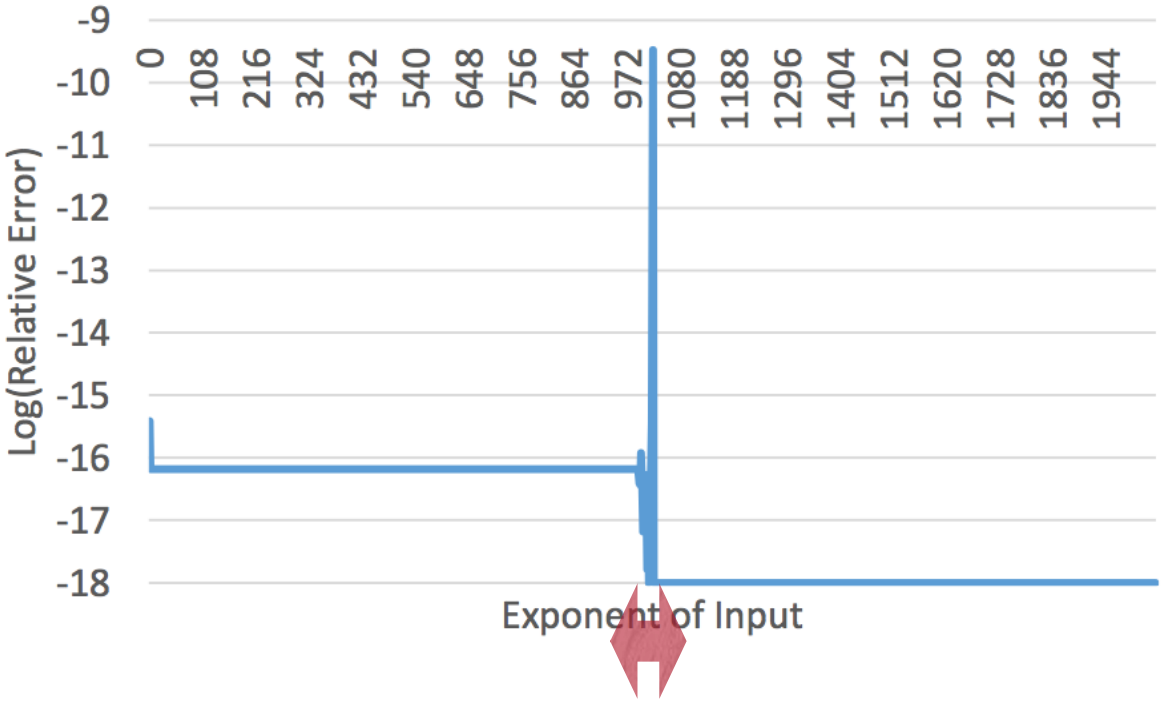



Fig. 1. erf at significand 0x34873b27b23c6

# 主要发现 3

在大误差的附近常常有高于平均误差的小波动。

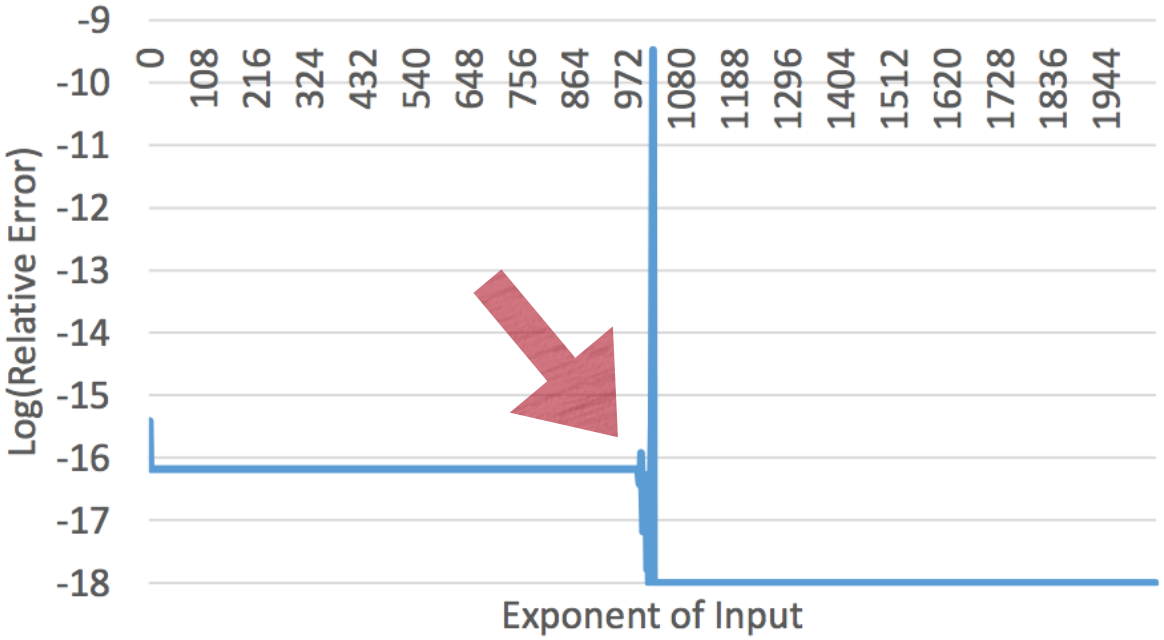



Fig. 1. erf at significand 0x34873b27b23c6

# 主要发现 4

大误差常常出现在浮点数中段位置。

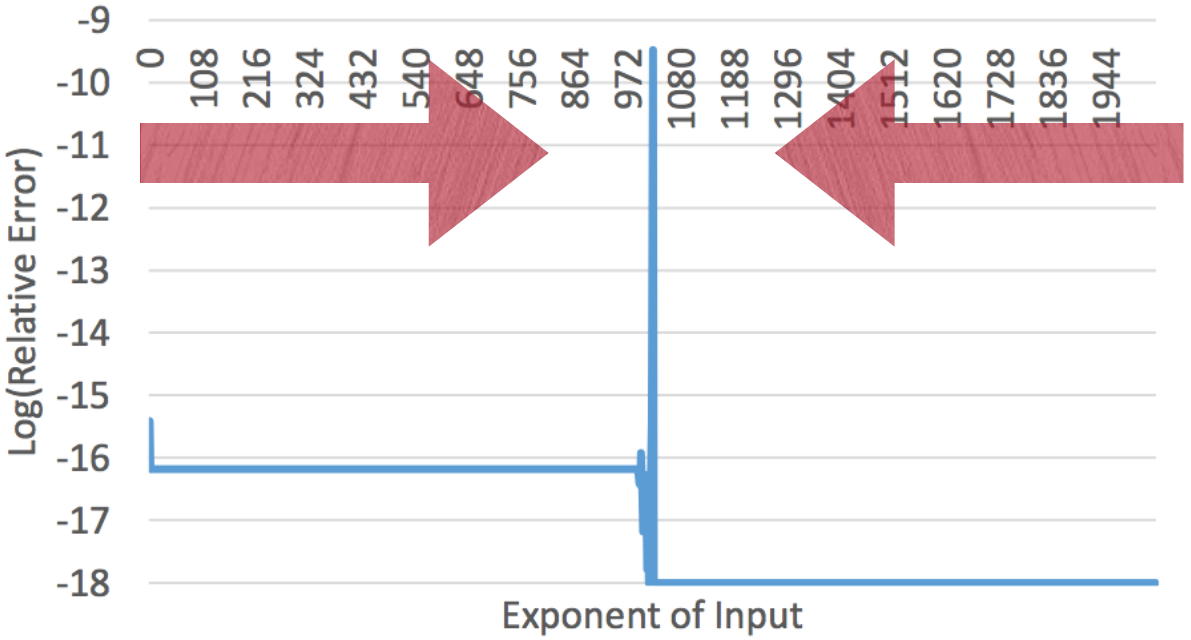



Fig. 1. erf at significand 0x34873b27b23c6

# 尾数位和浮点数之间的关系

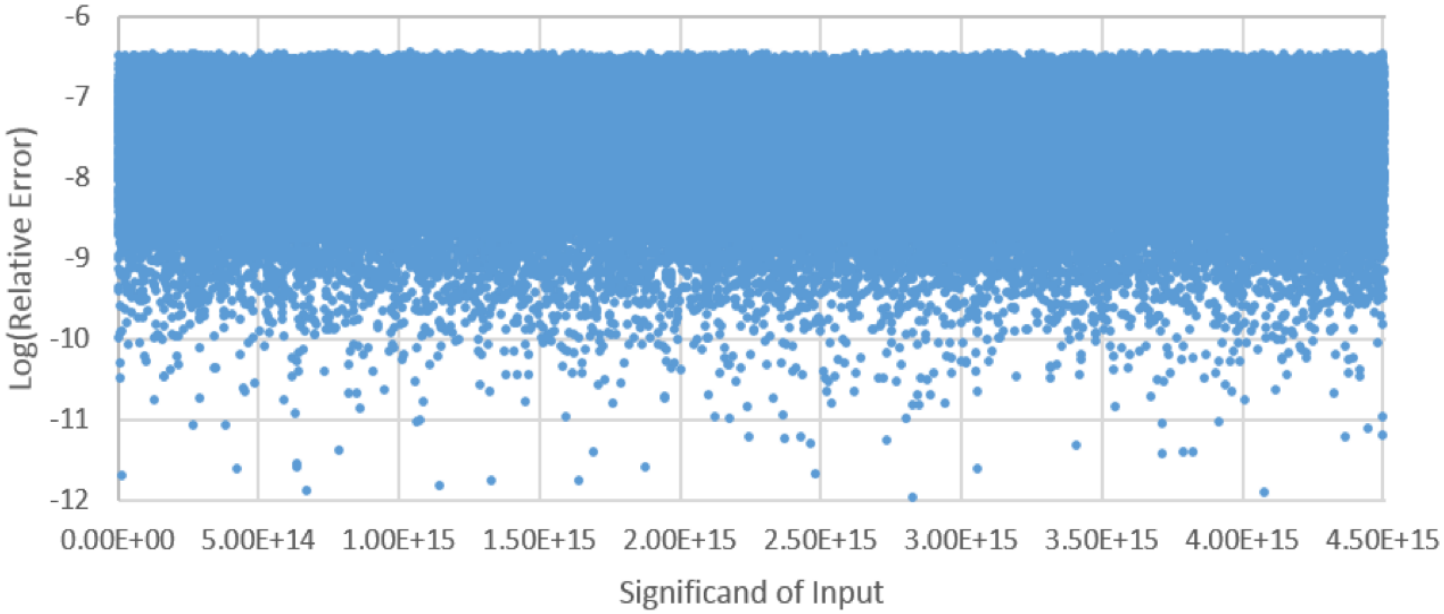



Fig. 2. erf at exponent 1023

# 主要发现 5

尾数位对浮点误差有显著影响。

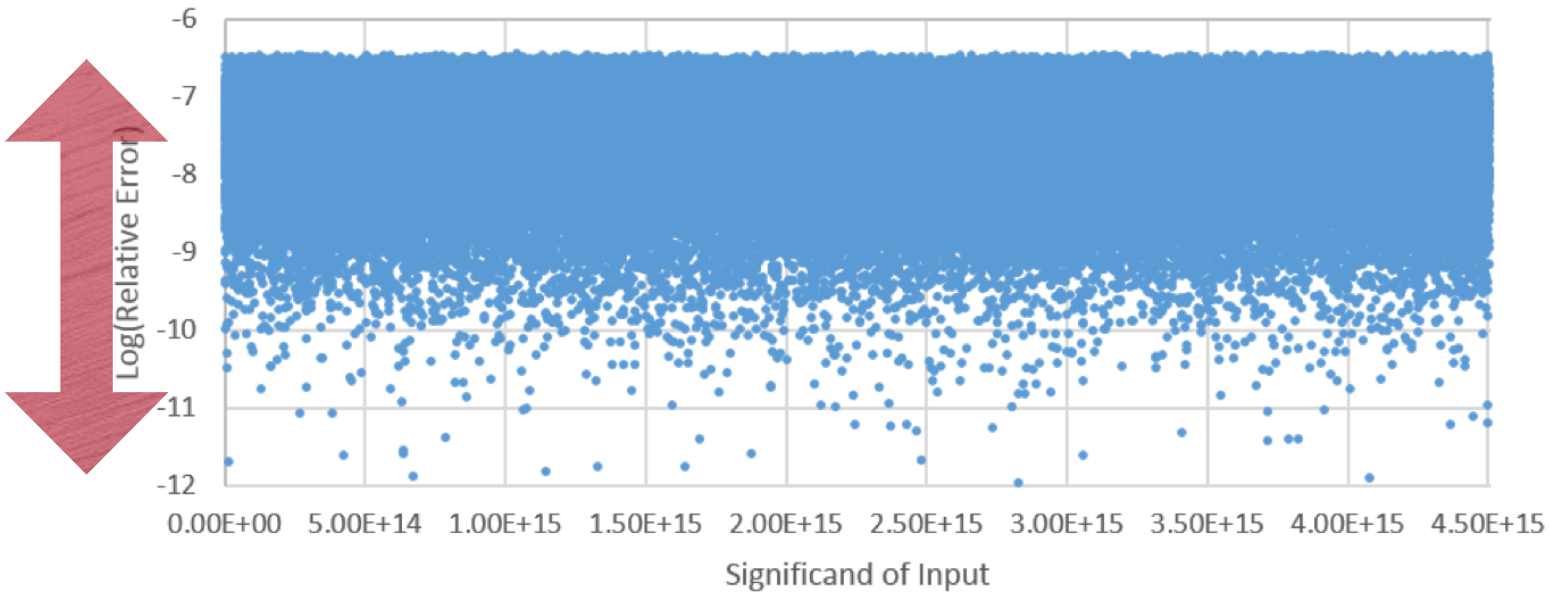



Fig. 2. erf at exponent 1023

# 主要发现 6

大量尾数都能触发大误差，并且在数轴上呈均匀分布。

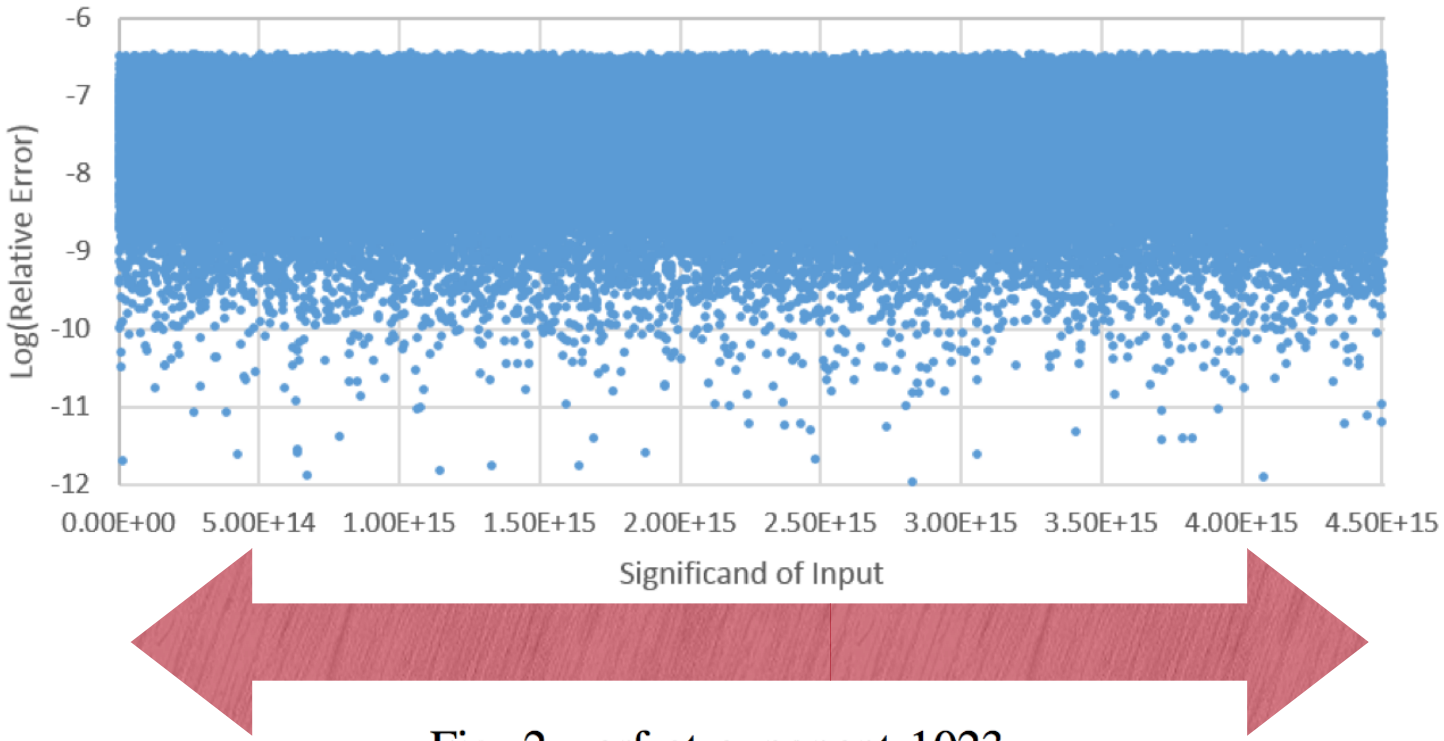



Fig. 2. erf at exponent 1023

# 局部敏感型遗传算法

- 根据以上特征，本研究设计了局部敏感型遗传算法。
- 对于指数位
  - 初始化时更多生成中位数附近。
  - 数值变异。
- 对于尾数位
  - 随机生成
  - 随机变异

# 实验评估

- 从GSL中选取154个函数。
- 与随机算法，标准遗传算法对比。

# 实验评估

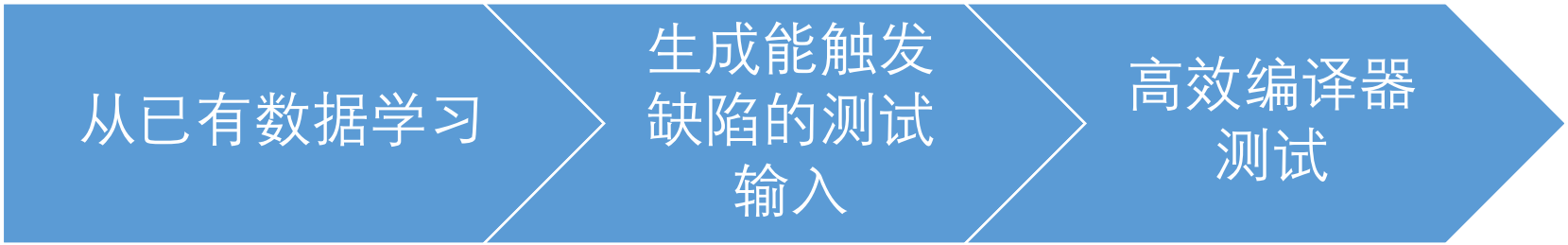
检测到最大误差

| Total | RAND | STD | LSGA | Tied |
|---|---|---|---|---|
| 154 | 11 (7%) | 24 (16%) | 105 (68%) | 14 (9%) |

Sign Test 结果

| | $n_+$ | $n_-$ | $N$ | $p$ |
|---|---|---|---|---|
| **LSGA vs. RAND** | 127 | 12 | 139 | < 4.14e-22 |
| **LSGA vs. STD** | 110 | 30 | 140 | < 2.46e-11 |
| **STD vs. RAND** | 93 | 40 | 133 | < 6.52e-06 |

# 查找缺陷：基于学习的编译器测试




[ICSE16,ICSE17,TSE18]

编译器测试效率提高17.16%-82.51%

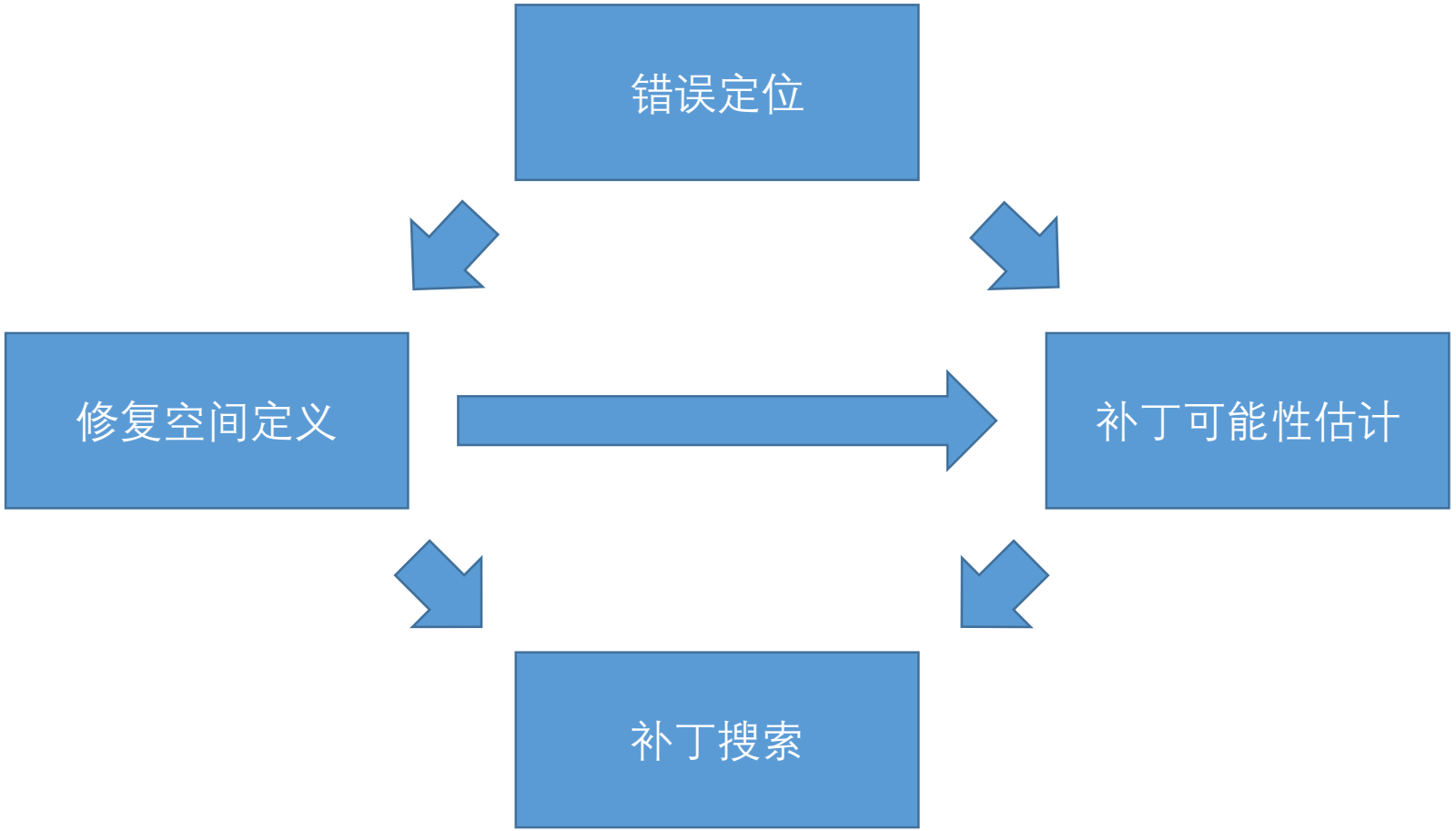
# 修复缺陷：
# 统计缺陷修复

# 缺陷自动修复定义

输入：一个程序和其正确性约束，并且程序不满足正确性约束
输出：一个补丁，可以使程序满足约束

研究和实践中考虑最广泛的正确性约束——
软件项目中的测试

# 缺陷修复四大支柱技术



# 典型缺陷修复技术: GenProg

错误定位

基于频谱的错误定位

修复空间定义

以下三种操作的组合:

- 在错误语句前插入一条同项目任意语句
- 将错误语句替换成同项目任意语句
- 删除任意语句

补丁可能性估计

通过测试越多越有可能

补丁搜索

遗传算法

# 缺陷修复三大挑战

- 正确率：由于测试的不完备性，通过测试的缺陷未必是正确的，导致缺陷修复技术很难达到较高的准确率
- 召回率：由于补丁的多样性，很难定义出准确的修复空间，或者在修复空间中很难定位到准确的补丁。
- 修复效率：目前技术修复一个缺陷往往需要数小时

“(First) open challenge.”
-- Claire Le Goues (CMU), ESEC/FSE, 2015

“Key discussion topic”
-- Dagstuhl Report 17022 “Automated Program Repair”

关于正确率

# 北京大学的近期工作

错误定位

集成错误定位方法
大幅提升错误定位正确率
[TSE19]

修复空间定义

通过分析历史补丁和项目代码来准确刻画修复空间
显著提升修复召回率
[ISSTA18]

补丁可能性估计

通过将生成过程分解来应用机器学习
显著提升修复正确率
[ICSE17(引用第一),GI18,ICSE18(引用第三)]

通过计算重用减少重复计算
显著提升修复效率
[ISSTA17(Distinguished Paper)]

补丁搜索

# 北京大学的近期工作

错误定位

集成错误定位方法
大幅提升错误定位正确率
[TSE19]

修复空间定义

通过分析历史补丁和项目代码来准确刻画修复空间
显著提升修复召回率
[ISSTA18]

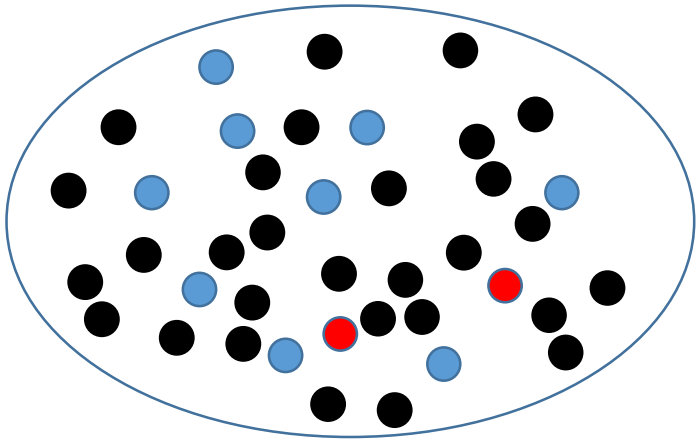
补丁可能性估计

通过将生成过程分解来应用机器学习
显著提升修复正确率
[ICSE17(引用第一),GI18,ICSE18(引用第三)]

通过计算重用减少重复计算
显著提升修复效率
[ISSTA17(Distinguished Paper)]

补丁搜索

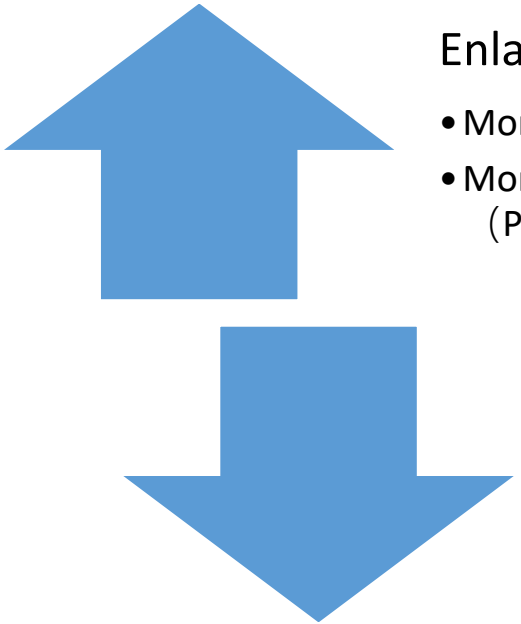
# Typical Repair Space



Patches failing the spec

Wrong patches meeting the spec

Correct patches



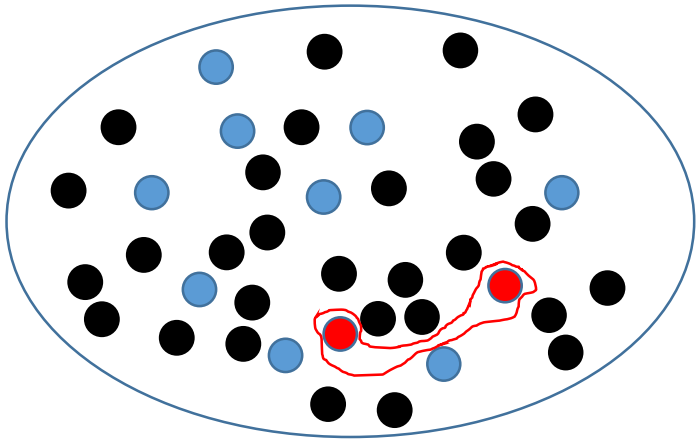
## Enlarge the space

- More likely to contain correct patches （Recall ↑↑）
- More difficult to locate the correct patches （Precision ↓↓Recall ↓）

## Shrink the space

- Less likely to contain correct patches （Recall ↓↓）
- Easier to locate the correct patches （Precision ↑↑Recall ↑）

# Ideal Repair Space



Patches failing the spec

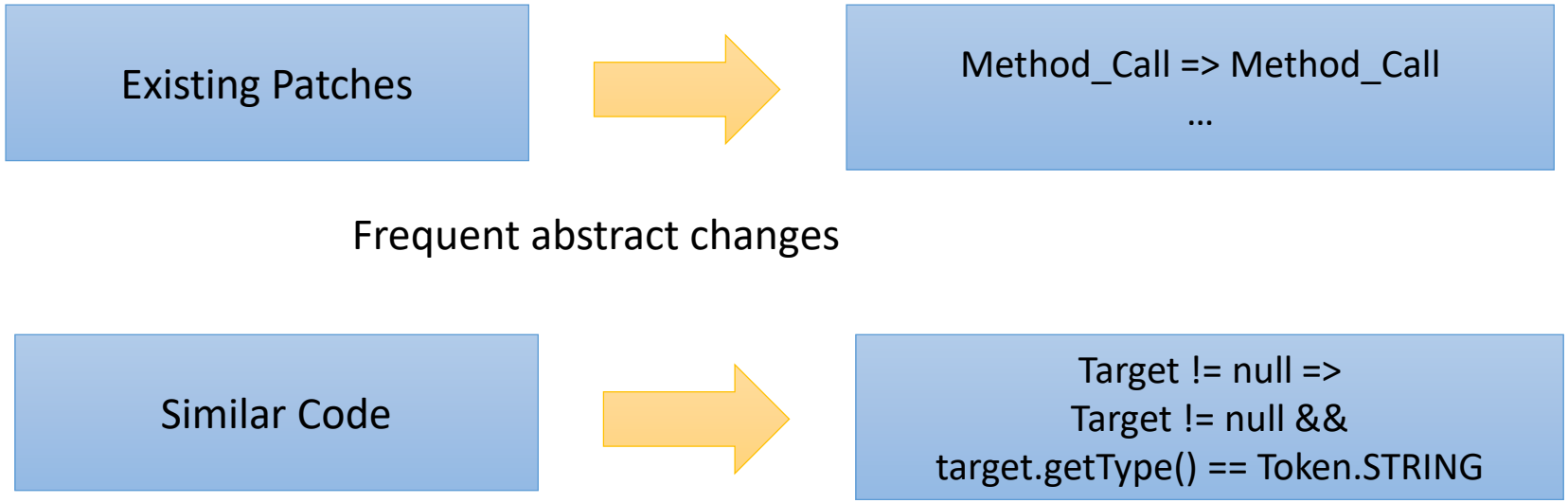
Wrong patches meeting the spec

Correct patches

- Containing the correct patches
- Containing no wrong patches, especially those meeting the specification

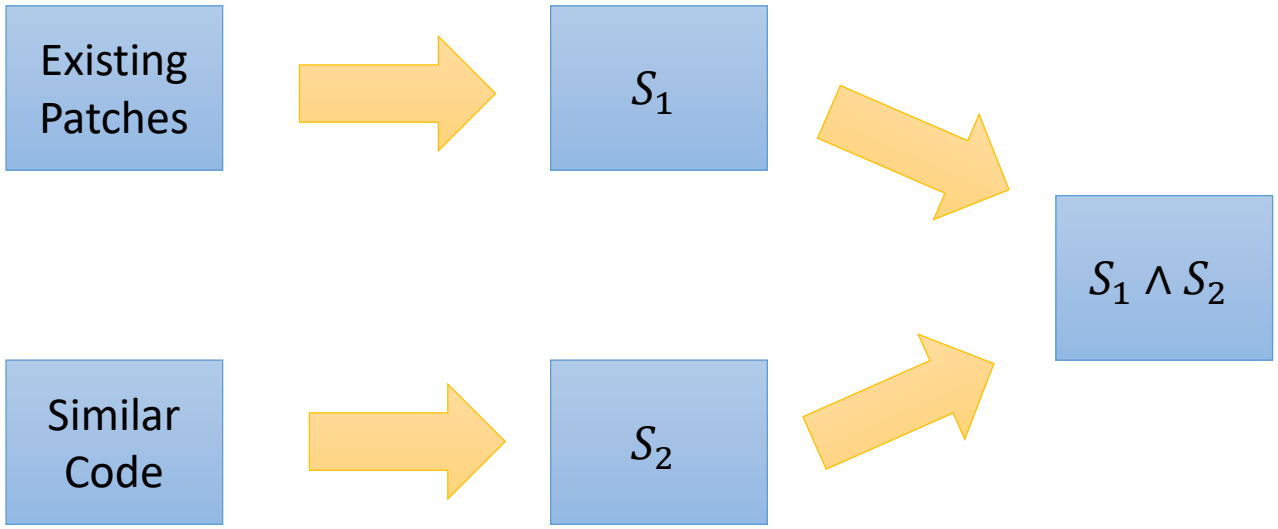
# SimFix

- Characterizing the space from two sources



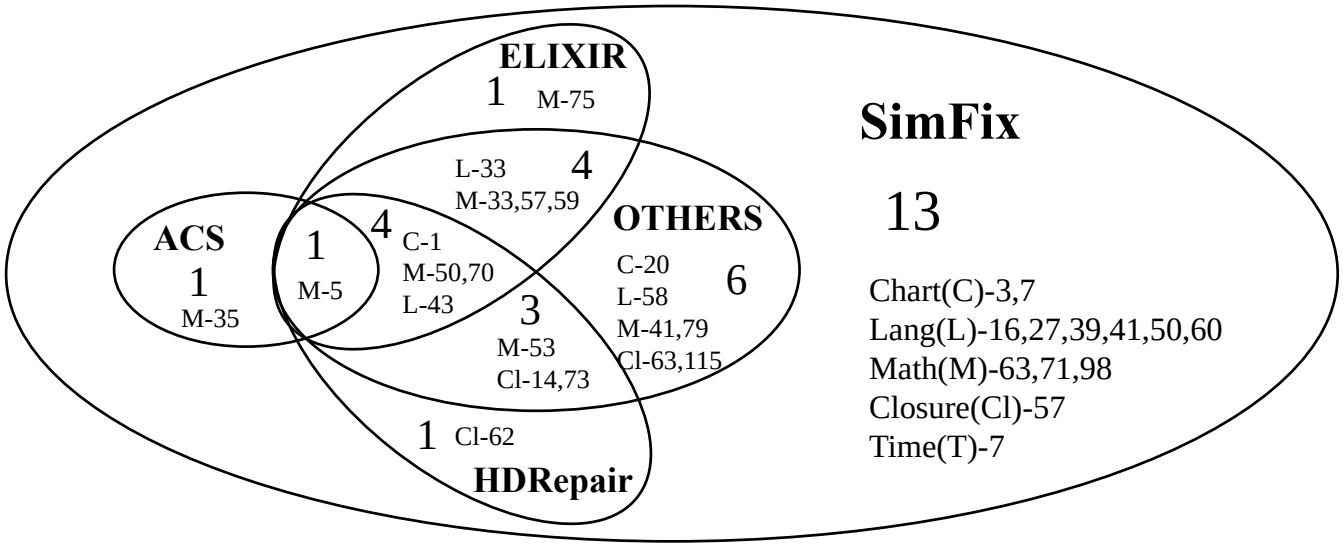
# SimFix

- Characterizing the space from two sources



# SimFix——Evaluation

- **Recall**: Fix 34 defects on Defects4J, including 13 uniquely repaired bugs, highest among all methods
- **Precision**: 60.7%




Bugs fixed by SimFix and related approaches

# 北京大学的近期工作

错误定位

集成错误定位方法
大幅提升错误定位正确率
[TSE19]

修复空间定义

通过分析历史补丁和项目代码来准确刻画修复空间
显著提升修复召回率
[ISSTA18]

补丁可能性估计

通过将生成过程分解来应用机器学习
显著提升修复正确率
[ICSE17(引用第一),GI18,ICSE18(引用第三)]

通过计算重用减少重复计算
显著提升修复效率
[ISSTA17(Distinguished Paper)]

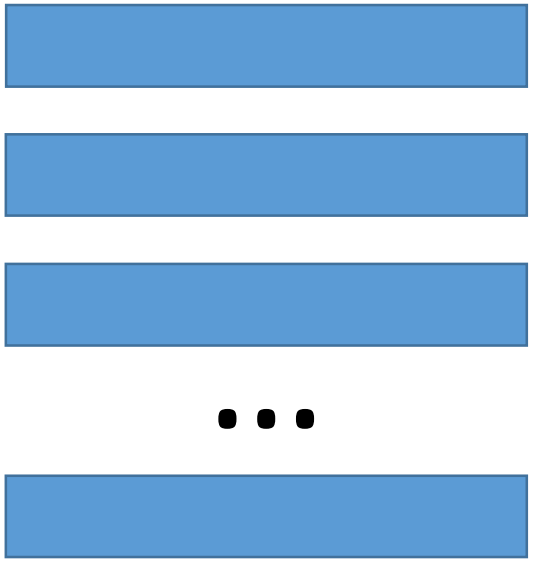
补丁搜索

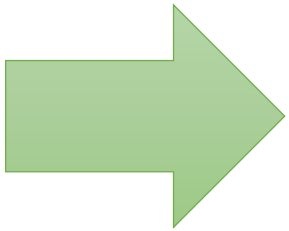
# Estimating Patch Probability

- Existing work: designing heuristic rules
  - passed tests/syntactic rules/semantic rules
  - Limited effectiveness
- Existing work: applying machine learning
  - Cannot applied to too large spaces
  - Solution 1: apply to abstract space
    - Too coarse-grained
  - Solution 2: using simple models with few features
    - Low accuracy

# Idea: Divide and Conquer

- Structurally decompose the search space





First component:
50 choice

Second component:
50 choice

Third component:
40 choice

Original Space:
10000 elements

# Learning to Synthesize

- A framework combining four tools
  - **Rewriting Rules**: defining a search problem
  - **Constraint Solving**: pruning off invalid choices in each step
  - **Machine Learning**: estimating the probabilities of choices in each step
  - **Search algorithms**: solving the search problem

# Application – Repairing incorrect conditions

- Condition bugs are common

```
   hours = convert(value);
+ if (hours > 12)
+   throw new ArithmeticException();
```

Missing boundary checks

```
-  if (hours >= 24)
+ if (hours > 24)
      withinOneDay=true;
```

Conditions too weak or too strong

- Existing work can pinpoint faulty conditions
- Generate a new condition to replace a faulty one

# Evaluation Results

- **Recall**: Fix 28 defects on 224 bugs from Defects4J, including 8 uniquely repaired bugs
  - 64.7% more fixed bugs than ACS, our previous work on repairing conditional bugs
- **Precision**: 76%

| Proj. | Total | ConCap | ACS | Nopol | SimFix | SKETCHFIX | CapGen |
|---|---|---|---|---|---|---|---|
| Chart | 26 | 3 (3) | 2 (0) | 1 (5) | 4 (4) | 6 (2) | 4 (0) |
| Math | 106 | 19 (6) | 12 (4) | 1 (20) | 14 (12) | 7 (1) | 12 (4) |
| Lang | 65 | 4 (0) | 2 (2) | 3 (4) | 9 (3) | 3 (1) | 5 (0) |
| Time | 27 | 2 (0) | 1 (0) | 0 (1) | 1 (0) | 0 (1) | 0 (0) |
| **Total** | 224 | 28 (9) | 17 (6) | 5 (30) | 28 (19) | 16 (5) | 21 (4) |
| **Precision** | - | 76% | 74% | 14% | 60% | 76% | 84% |
| **Recall** | - | 13% | 8% | 2% | 13% | 7% | 9% |

# 北京大学的近期工作

错误定位

集成错误定位方法
大幅提升错误定位正确率
[arXiv:1803.09939]

修复空间定义

通过分析历史补丁和项目代码来准确刻画修复空间
显著提升修复召回率
[ISSTA18]

补丁可能性估计

通过将生成过程分解来应用机器学习
显著提升修复正确率
[ICSE17(引用第二多),GI18,ICSE18]

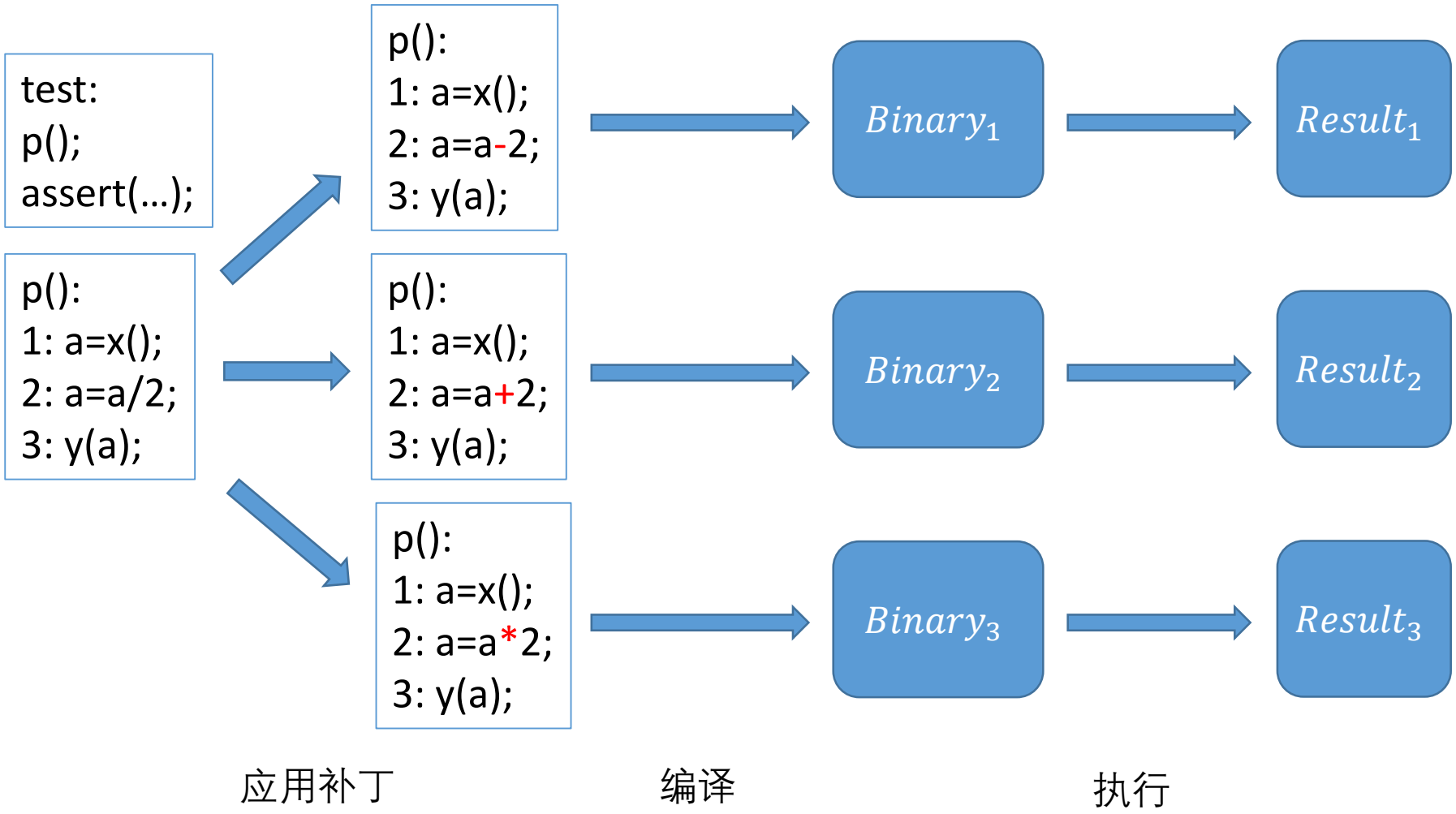
通过计算重用减少重复计算
显著提升修复效率
[ISSTA17(Distinguished Paper)]

补丁搜索

# 补丁搜索

- 现有缺陷修复技术常常需要花费数小时搜索补丁
- 补丁确认：补丁搜索过程中最耗时的部分
  - 应用补丁，检查是否所有测试都通过
- 能否加速该过程？

# 补丁确认过程


test:
p();
assert(…);
p():
1: a=x();
2: a=a/2;
3: y(a);
p():
1: a=x();
2: a=a-2;
3: y(a);
p():
1: a=x();
2: a=a+2;
3: y(a);
p():
1: a=x();
2: a=a*2;
3: y(a);
$Binary_1$
$Binary_2$
$Binary_3$
$Result_1$
$Result_2$
$Result_3$
应用补丁
编译
执行

# 编译的冗余

```
p():
1: a=x();
2: a=a-2;
3: y(a);
x():
…
y():
…
```

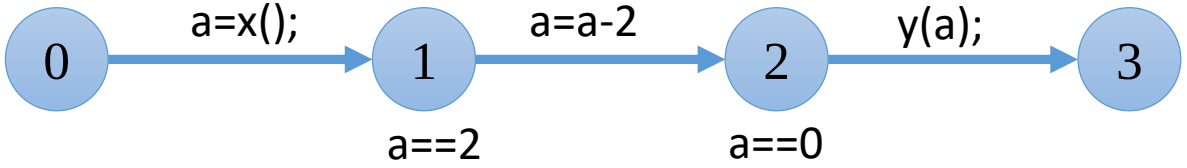
```
p():
1: a=x();
2: a=a+2;
3: y(a);
x():
…
y():
…
```

```
p():
1: a=x();
2: a=a*2;
3: y(a);
x():
…
y():
…
```

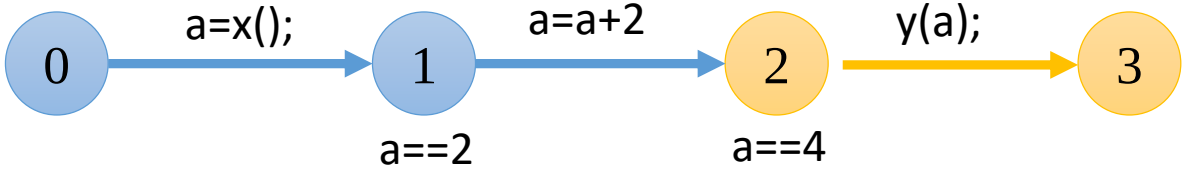
- x()和y()在三个版本中完全一样，但是被编译了三次

# 运行的冗余

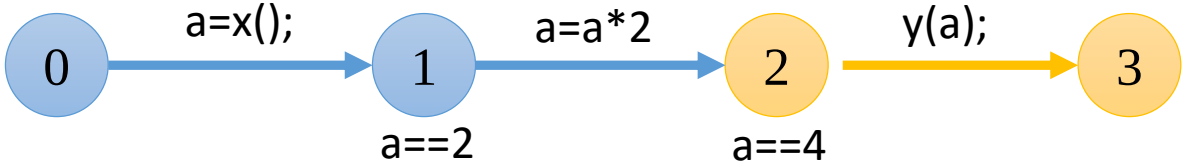
```
1: a=x();
2: a=a-2;
3: y(a);
```



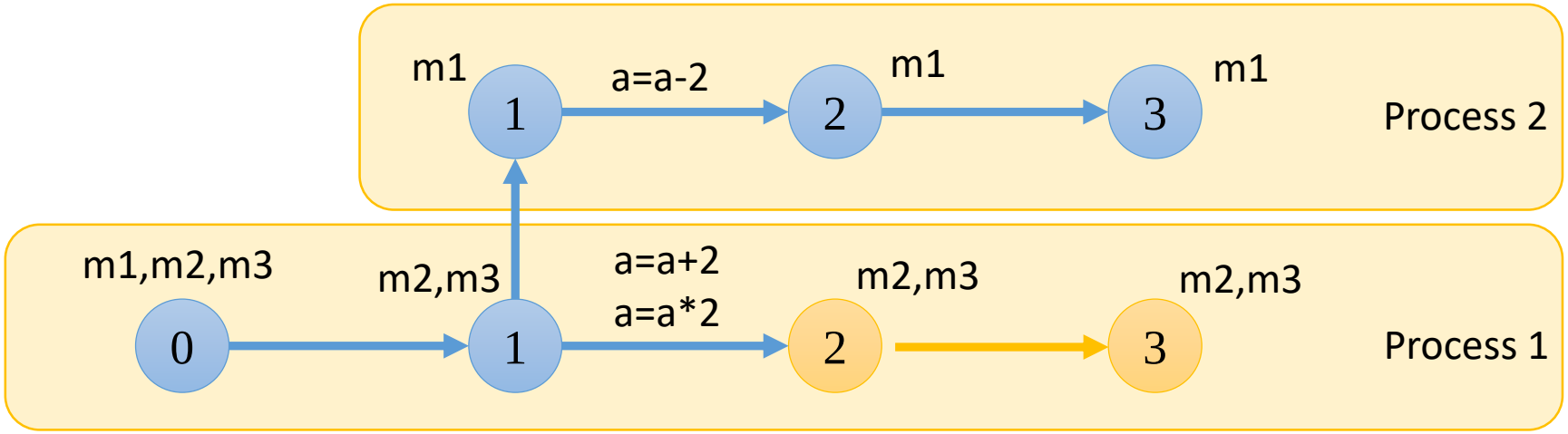

```
1: a=x();
2: a=a+2;
3: y(a);
```




```
1: a=x();
2: a=a*2;
3: y(a);
```



# 我们的工作：AccMut

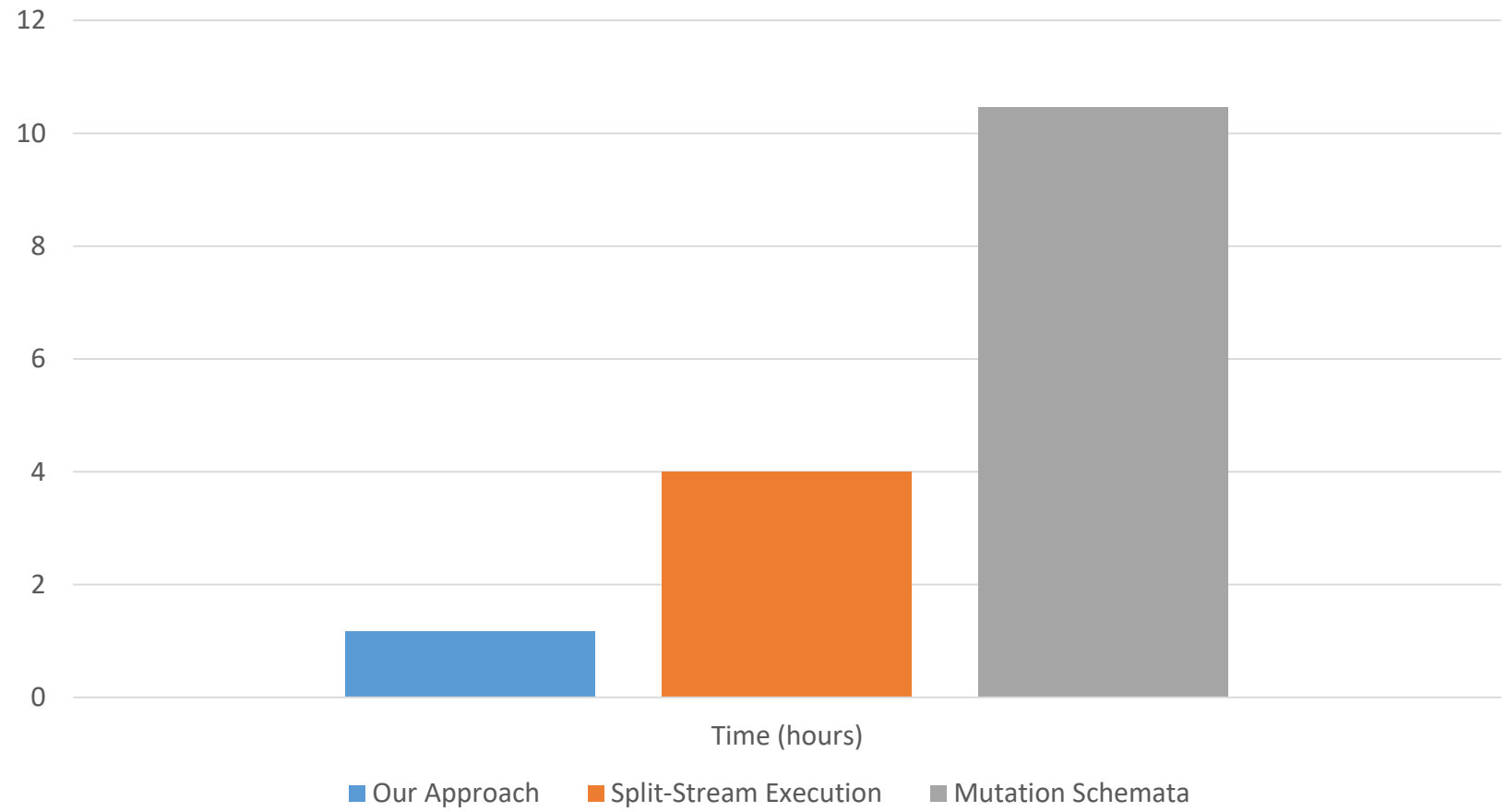



- 将所有补丁编译成一个超程序
- 超程序从一个进程开始执行，代表所有补丁
- 碰见有不同之处，超程序检查并分类不同语句的执行结果
- 对于每个等价结果类创建一个进程执行

# 如何实现超程序

- 需要保证 额外开销 << 去掉的重复计算
- 降低等价状态检查的额外开销
  - 运用抽象解释技术，将检查映射到一个开销较小的抽象域进行
- 降低结果分类的开销
  - 通过设计算法，使得分类算法依赖于一些上限较低的项

# 方法效果




在变异上模拟的效果：相比已有方法，我们比SSE有2.56倍加速，比MS有8.95倍加速

# 和华为的合作

- 一期：内存泄漏修复工具
- 二期：统计缺陷修复
- 非正式合作：编译器测试加速技术
- 期待更多合作!