My optimizations on cflow's non-determinism and overtainting

Presented by Wen Fan, USTC

Nov xth, 2021

Content

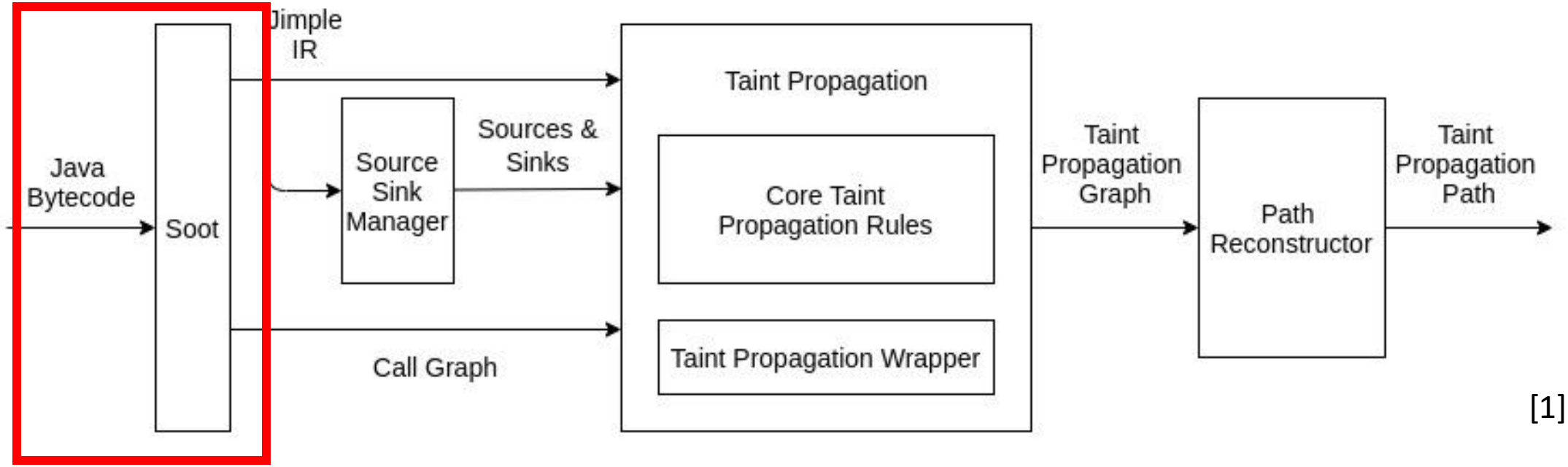
- Review
- Problem
- Solution
- Evaluation
- Conclusion

Content

- Review
- Problem
- Solution
- Evaluation
- Conclusion

Review

- cflow is a static taint analysis tool for Java application.
- It has 3 steps:



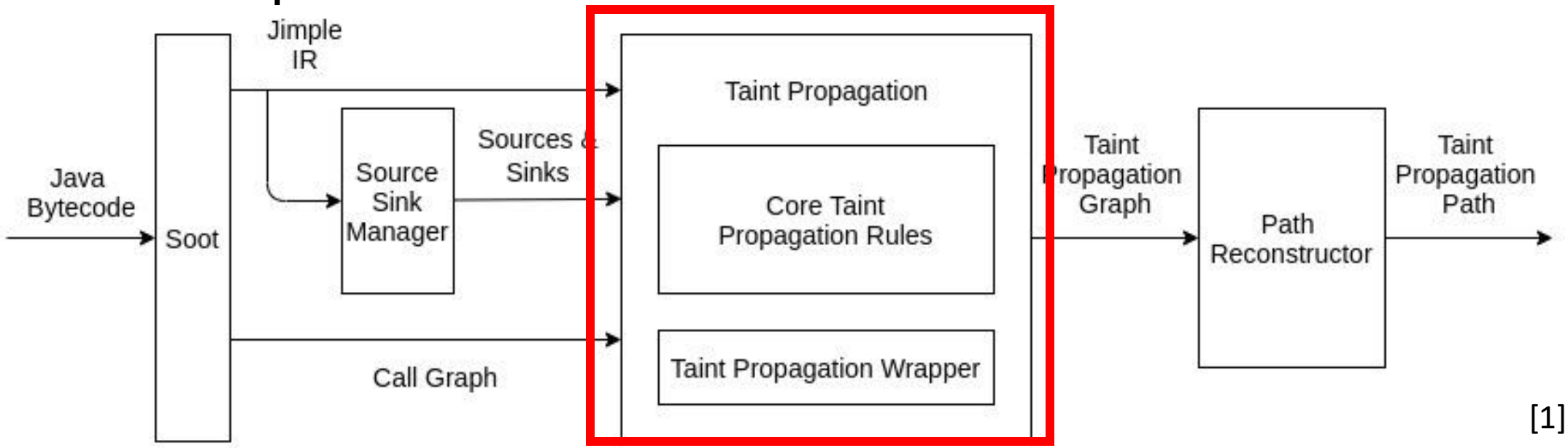
[1]

- Step 1: Get Jimple IR and call graph from Java bytecode

[1] Zhanghao Chen, Zijie Lu cFlow: Flow-based Configuration Analysis Framework for Java Bytecode Based Cloud Systems

Review

- cflow is a static taint analysis tool for Java application.
- It has 3 steps:



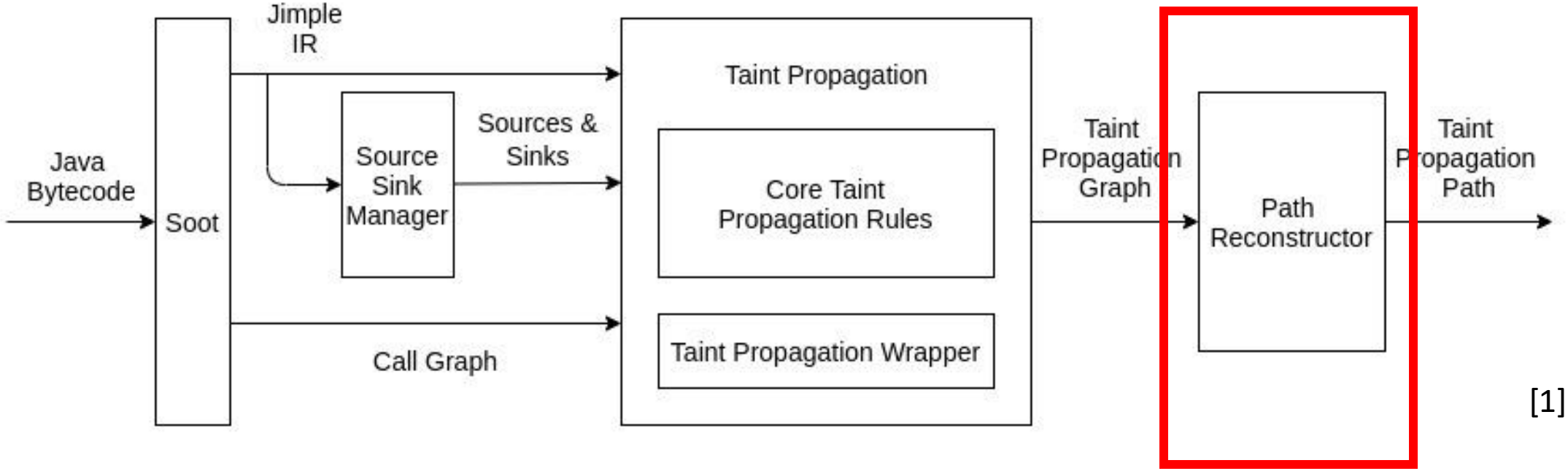
[1]

- Step 2: Analyze Jimple IR to detect and propagate taints

[1] Zhanghao Chen, Zijie Lu cFlow: Flow-based Configuration Analysis Framework for Java Bytecode Based Cloud Systems

Review

- cflow is a static taint analysis tool for Java application.
- It has 3 steps:



[1]

- Step 3: Run DFS to reconstruct taint propagation path

[1] Zhanghao Chen, Zijie Lu cFlow: Flow-based Configuration Analysis Framework for Java Bytecode Based Cloud Systems

Content

- Review
- Problem
- Solution
- Evaluation
- Conclusion

Problem - Overview

- 1. Non-deterministic output
  - Run cflow twice and get different results
- 2. Overtainting
  - False-positive

Problem – Non-deterministic output

- The sequence of taints added into successor is not deterministic.
  - e.g. run cflow twice, and successor is
  - 1st time: { stmt 1 stmt 2 } , 2nd time: { stmt 2 stmt 1 }

Problem – Non-deterministic output

- The sequence of taints added into successor is not deterministic.
  - e.g. run cflow twice, and successor is
  - 1st time: { stmt 1 stmt 2 } , 2nd time: { stmt 2 stmt 1 }

- The sorting cannot make successor deterministic.
  - e.g. after the sort, and successor is
  - 1st time: { stmt 1 stmt 2 } , 2nd time: { stmt 2 stmt 1 }

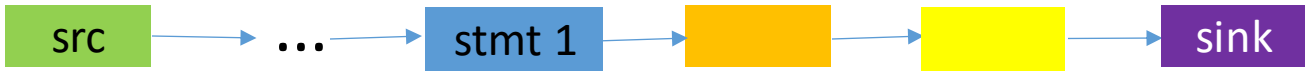
Problem – Non-deterministic output

- During path reconstruction, the exploration is non-deterministic:
  - 1st time: { stmt 1 stmt 2 }

  - 2nd time: { stmt 2 stmt 1 }

Problem – Non-deterministic output

- During path reconstruction, the exploration is non-deterministic.
  - 1st time: { stmt 1 stmt 2 }



  - 2nd time: { stmt 2 stmt 1 }

- So they get different taint propagation paths.

Problem – Non-deterministic output

- The sequence of taints added into successor is not deterministic.
  - e.g. run cflow twice:
  - 1st time: { obj 1 obj 2 } , 2nd time: { obj 2 obj 1 } Why?

- There are two reasons:
  - 1. The sequence of intra-procedural analysis is not deterministic.

```
for (SootMethod sm : methodList) {
    // ...
    for (Taint entryTaint : entryTaints) {
        TaintFlowAnalysis analysis = new TaintFlowAnalysis(b,...,entryTaint,...);
        analysis.doAnalysis();
     }
}
```

Problem – Non-deterministic output

- The sequence of taints added into successor is not deterministic.
  - e.g. run cflow twice:
  - 1st time: { obj 1 obj 2 } , 2nd time: { obj 2 obj 1 } Why?

- There are two reasons:
  - 1. The sequence of intra-procedural analysis is not deterministic.

```
for (SootMethod sm : methodList) {
    // ...
    for (Taint entryTaint : entryTaints) {
        TaintFlowAnalysis analysis = new TaintFlowAnalysis(b,...,entryTaint,...);
        analysis.doAnalysis();
    }
}
```

entryTaints is a HashSet

The sequence of this iteration is not deterministic

Problem – Non-deterministic output

- The sequence of taints added into successor is not deterministic.
  - e.g. run cflow twice:
  - 1st time: { obj 1 obj 2 } , 2nd time: { obj 2 obj 1 } Why?

- There are two reasons:
  - 2. In intra-procedural analysis, the sequence of new taint generation is not deterministic.

```
for (Taint t : in) {
    //...
}
```

Problem – Non-deterministic output

- The sequence of taints added into successor is not deterministic.
  - e.g. run cflow twice:
  - 1st time: { obj 1 obj 2 } , 2nd time: { obj 2 obj 1 } Why?

- There are two reasons:
  - 2. In intra-procedural analysis, the sequence of new taint generation is not deterministic.

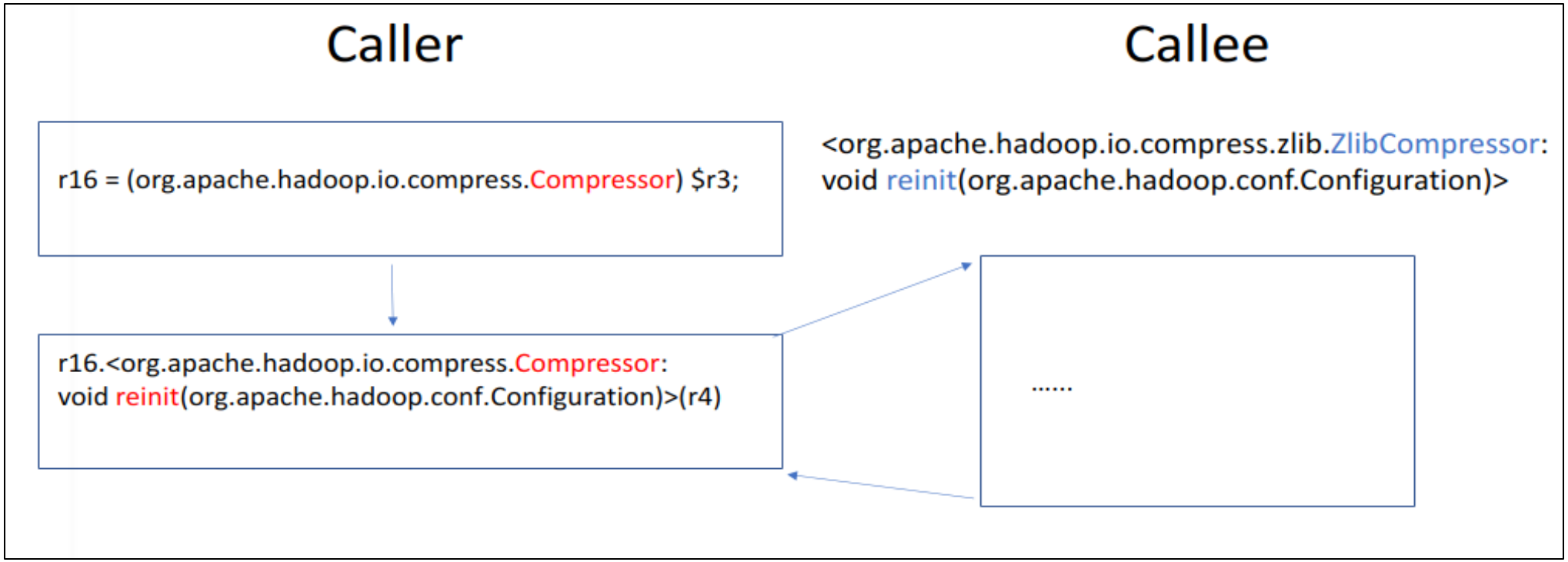
In is a HashSet

```
for (Taint t : in) {
    //...
}
```

The sequence of this iteration is not deterministic

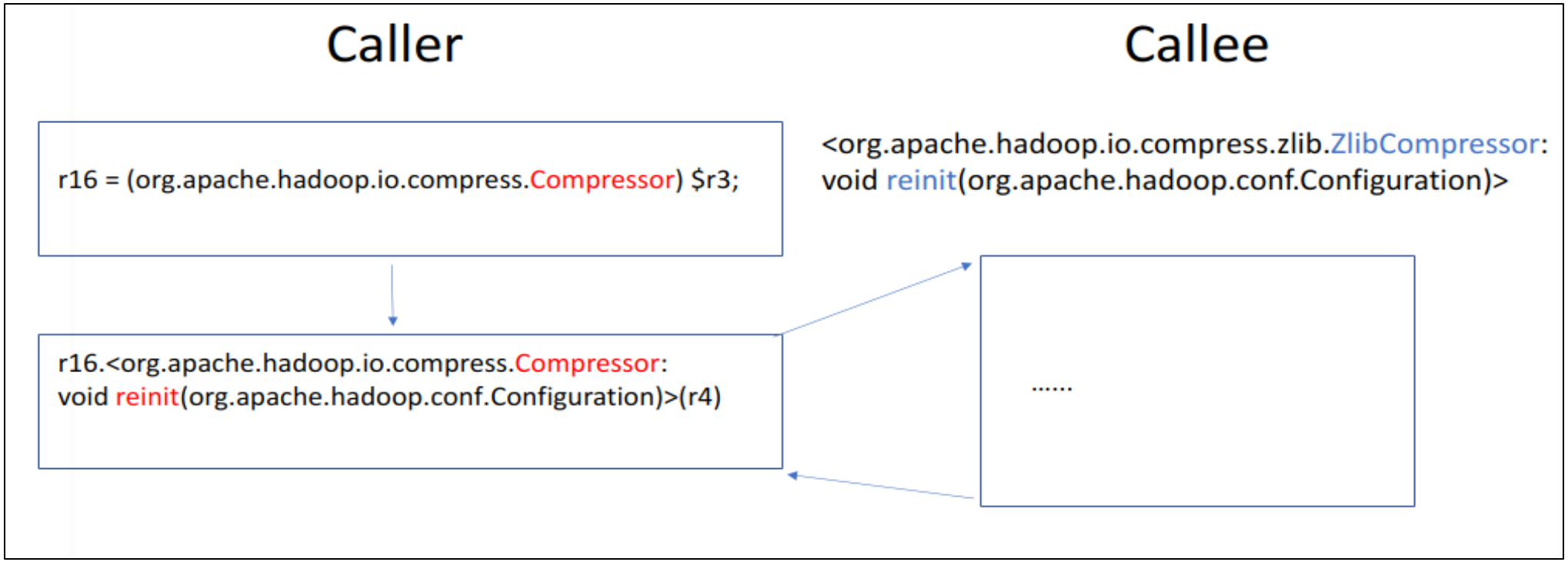
Problem – Overtainting

- cflow gets imprecise callee if polymorphism exists.
  - e.g. a case in Hadoop Common:



Problem – Overtainting

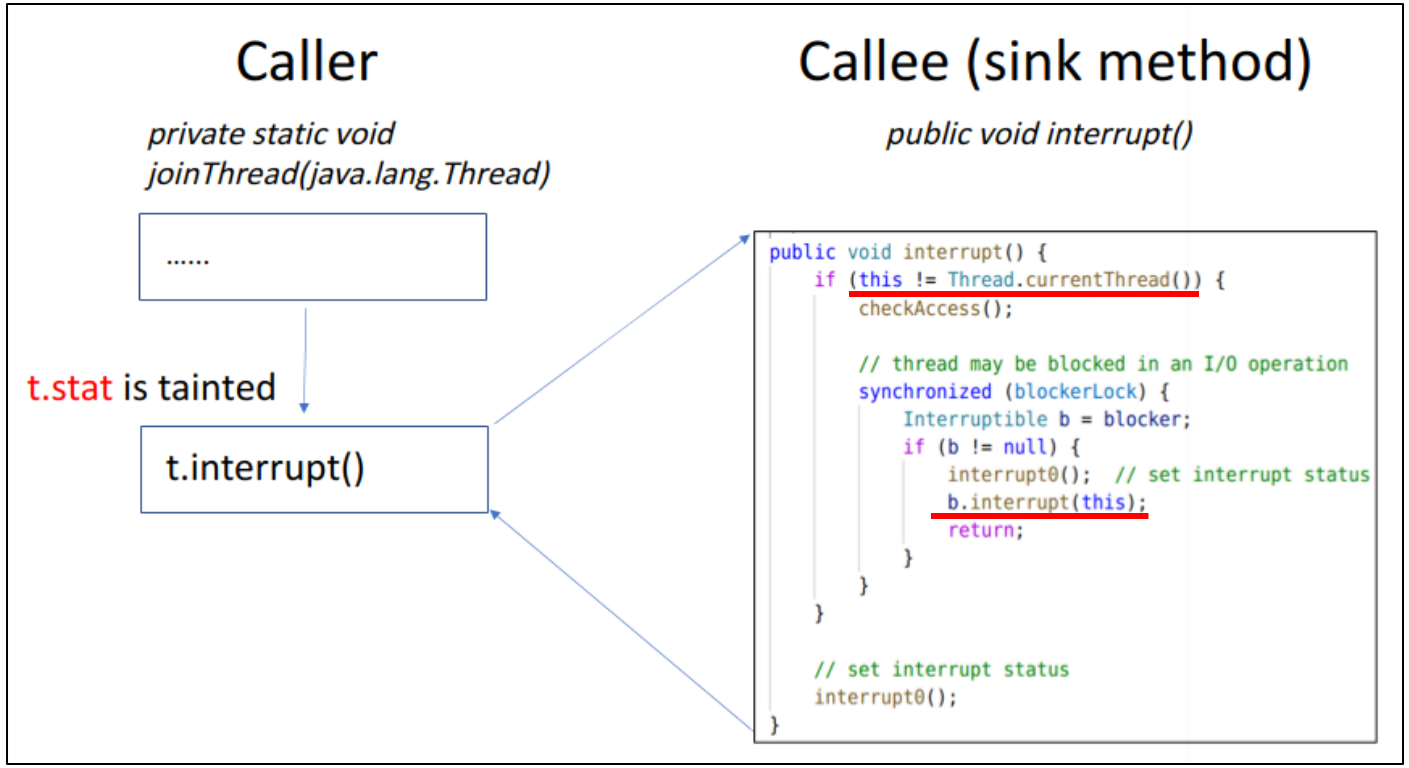
- cflow gets imprecise callee if polymorphism exists.
  - e.g. a case in Hadoop Common:



cflow doesn't know the type of object that r16 refers to.

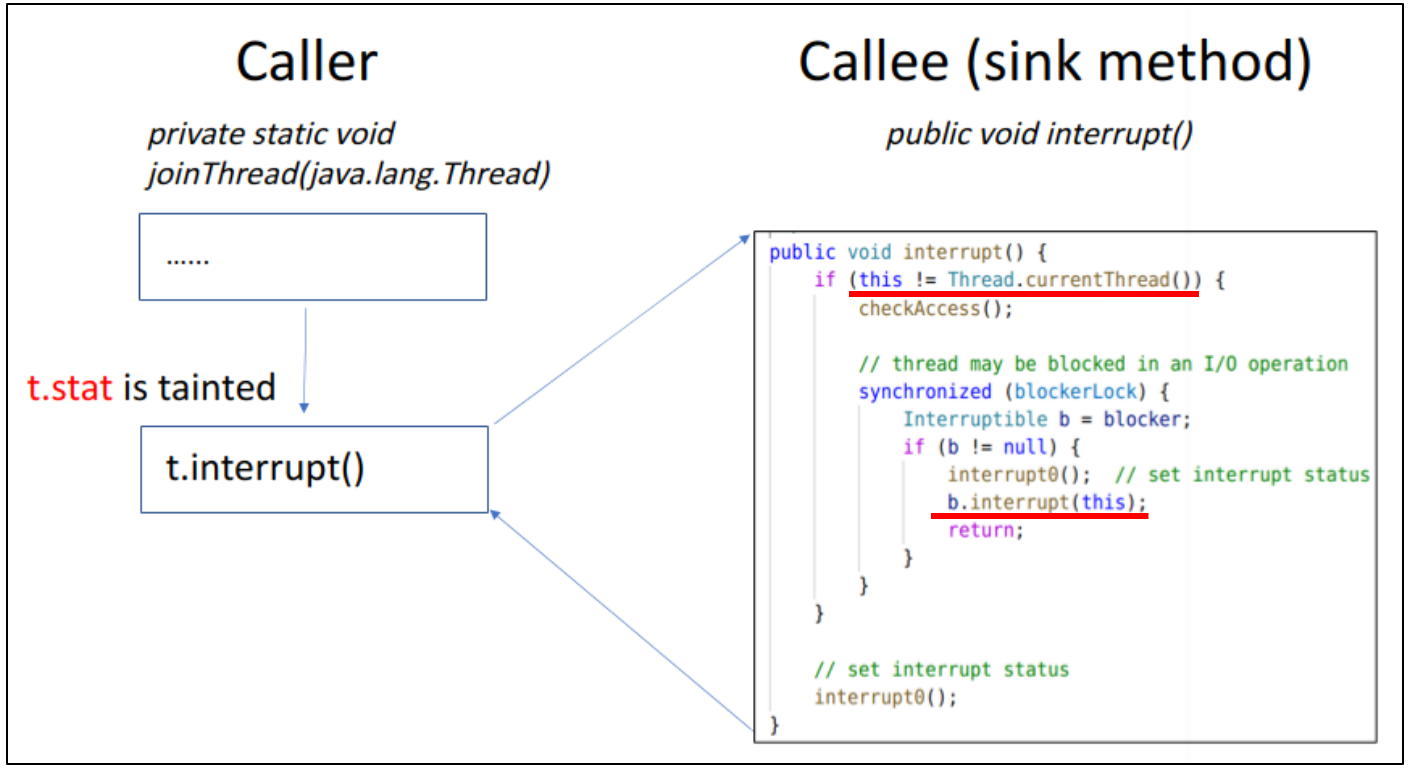
Problem – Overtainting

- cflow doesn't check whether the tainted object is used in sink.
  - e.g. a case in Hadoop Common:



Problem – Overtainting

- cflow doesn't check whether the tainted object is used in sink.
  - e.g. a case in Hadoop Common:



Actually, t.stat may be used in b.interrupt(this), but cflow doesn't check that

Content

- Review
- Problem
- Solution
- Evaluation
- Conclusion

Solution - Overview

- For imprecise callee,
  - I implement a points-to analysis to get the runtime type of each object,
  - I then check the type of each base object to get a precise callee.

- At sink method,
  - I check whether the tainted object is used in sink.

Solution – Points-to analysis

- The points-to analysis is a (semi)context-, flow-, field-sensitive.
- It stores the location of each object at each program point.
  - e.g.

```
String foo(Object o1, Object o2) {
    o1 = o2;                    // 1
    this.f = bar();             // 2
    return bar();               // 3
}

String bar() {
    return new String();        // 4
}
```

Init: o1 => Loc1, o2 => Loc2

Solution – Points-to analysis

- The points-to analysis is a (semi)context-, flow-, field-sensitive.
- It stores the location of each object at each program point.
  - e.g.

```
String foo(Object o1, Object o2) {
    o1 = o2;                    // 1
    this.f = bar();             // 2
    return bar();               // 3
}

String bar() {
    return new String();        // 4
}
```

Init: o1 => Loc1, o2 => Loc2

1: o1 => Loc2, o2 => Loc2

Solution – Points-to analysis

- The points-to analysis is a (semi)context-, flow-, field-sensitive.
- It stores the location of each object at each program point.
  - e.g.

```
String foo(Object o1, Object o2) {
    o1 = o2;                    // 1
    this.f = bar();             // 2
    return bar();               // 3
}

String bar() {
    return new String();        // 4
}
```

Init: o1 => Loc1, o2 => Loc2

1: o1 => Loc2, o2 => Loc2

2: o1 => Loc2, o2 => Loc2
this.f => newLoc@2,4

Solution – Points-to analysis

- The points-to analysis is a (semi)context-, flow-, field-sensitive.
- It stores the location of each object at each program point.
  - e.g.

```
String foo(Object o1, Object o2) {
    o1 = o2;                    // 1
    this.f = bar();             // 2
    return bar();               // 3
}

String bar() {
    return new String();        // 4
}
```

Init: o1 => Loc1, o2 => Loc2

1: o1 => Loc2, o2 => Loc2

2: o1 => Loc2, o2 => Loc2
this.f => newLoc@2,4

3: o1 => Loc2, o2 => Loc2
this.f => newLoc@2,4
retVal => newLoc@3,4

Solution – Points-to analysis

- The points-to analysis is a (semi)context-, flow-, field-sensitive.
- It stores the location of each object at each program point.
- For performance, it builds a summary to avoid repetitious analysis

```
String foo(Object o1, Object o2) {
    o1 = o2;                    // 1
    this.f = bar();             // 2
    return bar();               // 3
}

String bar() {
    return new String();        // 4
}
```

Summary:
(1) o1 => o2
(2) this.f => new loc@2,4
(3) retVal => new loc@3,4

Solution – Field-use analysis

- I assume that this object and parameters must be used in a method.
  - e.g.

```
void foo(Object o1, Object o2) {
    // do something
}
```

Solution – Field-use analysis

- I assume that this object and parameters must be used in a method.
  - e.g.

```
void foo(Object o1, Object o2) {
    // do something
}
```

this, o1, o2 must be used in foo()

this.f, o1.g, o2.h may not be used in foo()

- So I just need to check the use of the tainted field.

Solution – Field-use analysis

- The field-use analysis checks whether the field reference is used in sink method.
- There are several types of use:
  - Must: field use is detected.

```
void sink1() {
    int i = this.f;
    System.out.println(i);
}
```

this.f must
be used in sink1()

Solution – Field-use analysis

- The field-use analysis checks whether the field reference is used in sink method.
- There are several types of use:
  - Must: field use is detected.

```
void sink1() {
    int i = this.f;
    System.out.println(i);
}
```

this.f must be used in sink1()

  - May: base object is used in a further method call.

```
void sink2() {
    this.foo();
}
```

this.f may be used in sink2()

Solution – Field-use analysis

- The field-use analysis checks whether the field reference is used in sink method.
- There are several types of use:
  - Must: field use is detected.

```
void sink1() {
    int i = this.f;
    System.out.println(i);
}
```

this.f must be used in sink1()

  - May: base object is used in a further method call.

```
void sink2() {
    this.foo();
}
```

this.f may be used in sink2()

  - Never: the field is never used.

```
void sink3() {
    return;
}
```

this.f is never used in sink3()

Solution – Field-use analysis

- A flow insensitive analysis can do!!!
- It can analyze further callee within a limit.

Solution – Field-use analysis

- A flow insensitive analysis can do!!!

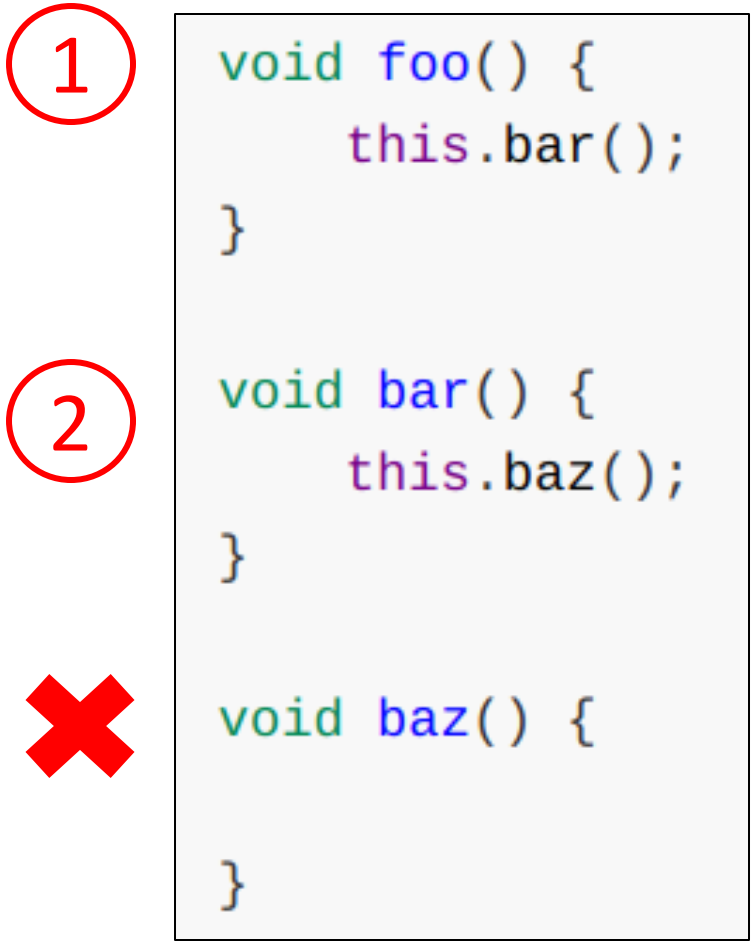
- It can analyze further callee within a limit.
  - e.g. check whether this.f is used in foo(),
    where max depth = 2.

```
void foo() {
    this.bar();
}

void bar() {
    this.baz();
}

void baz() {

}
```

Solution – Field-use analysis

- A flow insensitive analysis can do!!!
- It can analyze further callee within a limit.
  - e.g. check whether this.f is used in foo(), where max depth = 2.

this.f may be used in foo()



Solution – Field-use analysis

- A flow insensitive analysis can do!!!
- It can analyze further callee within a limit.
- Moreover, I build a summary to avoid reanalyzing the field use in the same method.

```
void foo() {
    // do something
    this.bar();
}

void bar() {
    // do something
    this.baz();
}

void baz(){
    // do something
}
```

Content

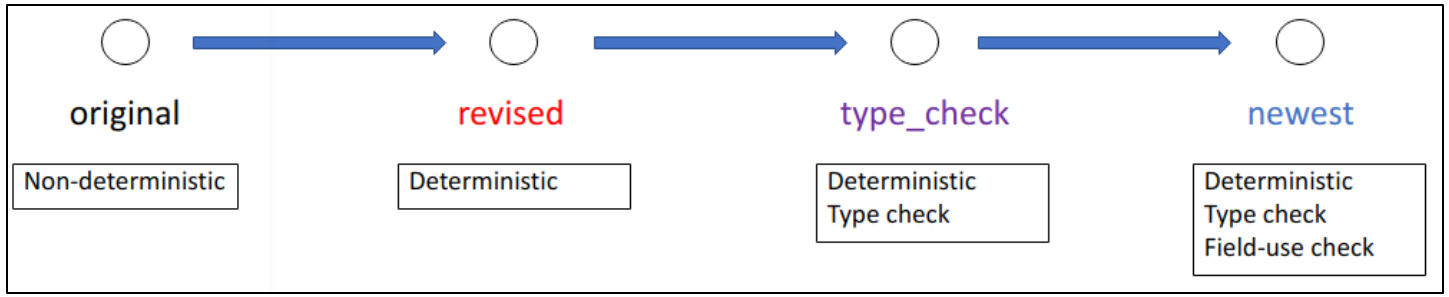
- Review
- Problem
- Solution
- Evaluation
- Conclusion

Evaluation – Questions

- Q1: Can cflow have deterministic output after the revision?
- Q2: How effective is points-to analysis?
- Q3: How effective is field-use analysis?
- Q4: What is the performance cost of the new implementation?

Evaluation – Configuration

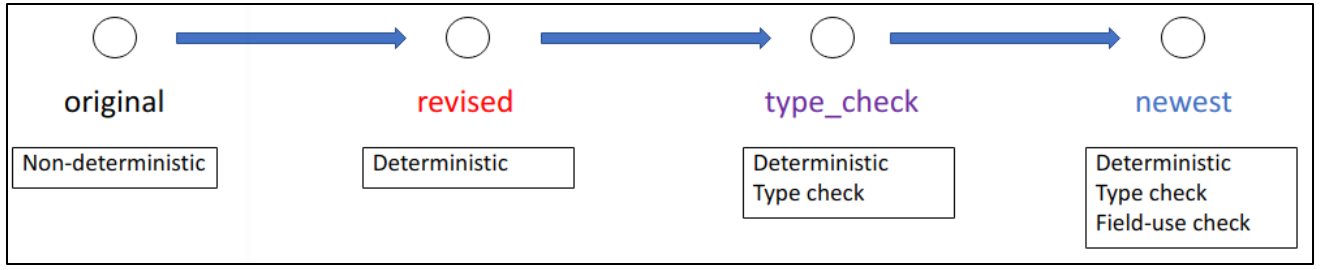
- Environment: A normal PC
  - CPU: 8 * Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
  - CPU cache: 6144 KB
  - Memory: 7845 MB
  - Swap area: 975 MB
  - hard disk: SAMSUNG MZVLW256HEHP-00000 SSD 236.26 GiB
  - OS: ubuntu 20.04
  - kernel: 5.4.0-84-generic
  - maven: 3.6.3
  - Java: 1.8.0_291
- On four versions of cflow



Evaluation – Q1

- Q1: Can cflow have deterministic output after the revision?

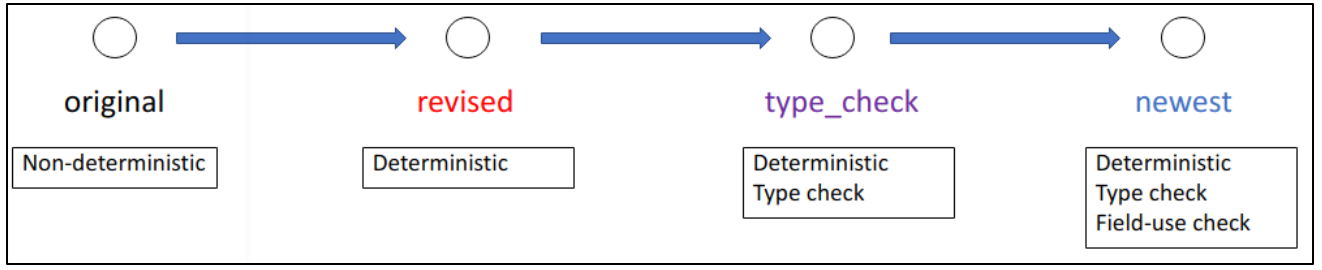
- Run each version on hadoop_common for 10 times, and
- Check whether difference occurs in the outputs of each version



Evaluation – Q1

- Q1: Can cflow have deterministic output after the revision?

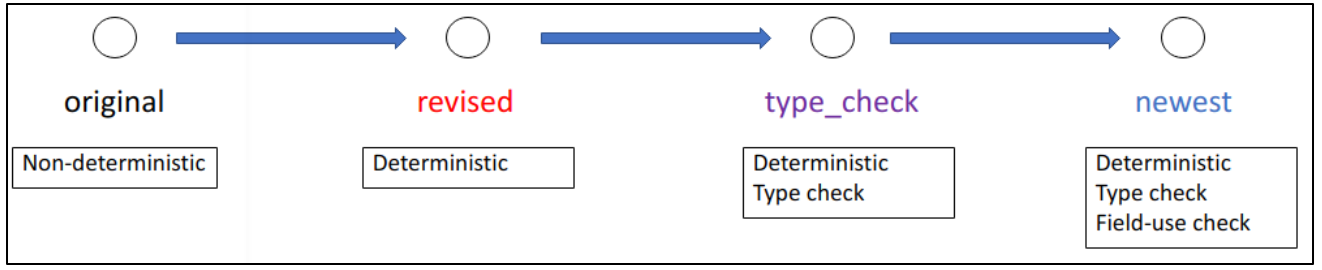
- Run each version on hadoop_common for 10 times, and
- Check whether difference occurs in the outputs of each version
  - Difference exists in the outputs of original
  - No difference exists in the outputs of revised, type_check, newest.



Evaluation – Q2

- Q2: How effective is points-to analysis?

- Run revised and type_check(with pta and w/o pta)
- Check diff between revised and type_check.

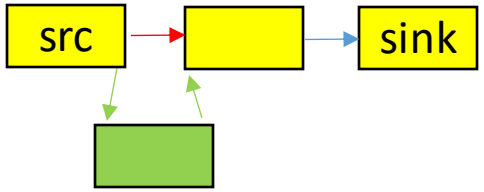


Evaluation – Q2

- Q2: How effective is points-to analysis?

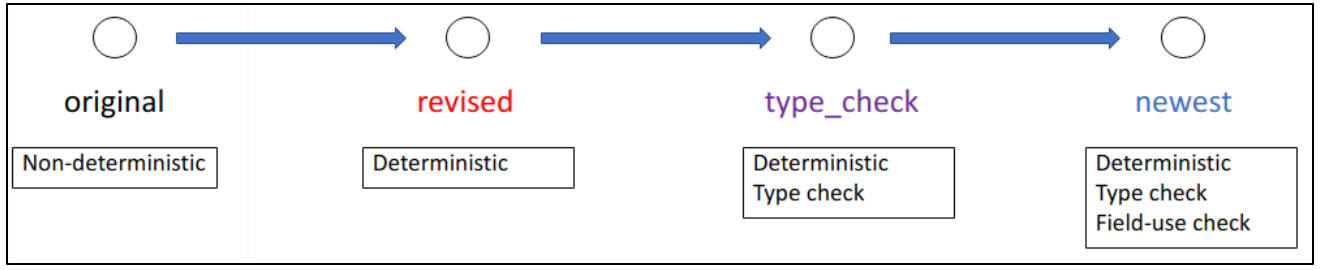
whole path:

sub path:



- Run revised and type_check(with pta and w/o pta)
- Check diff between revised and type_check.

add/remove	changes	without pta	with pta
remove	whole path	73	65
	sub path	3	3
add	whole path	0	0
	sub path	1	1

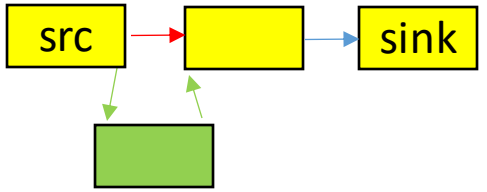


Evaluation – Q2

- Q2: How effective is points-to analysis?

whole path:

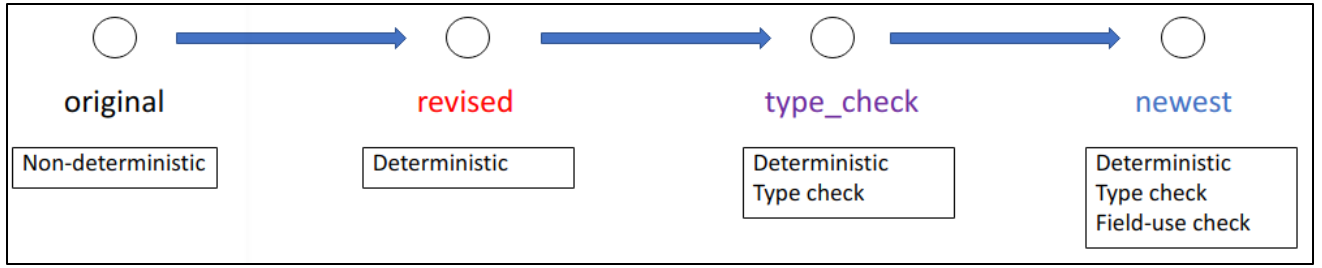
sub path:



- Run revised and type_check(with pta and w/o pta)
- Check diff between revised and type_check.

add/remove	changes	without pta	with pta
remove	whole path	73	65
	sub path	3	3
add	whole path	0	0
	sub path	1	1

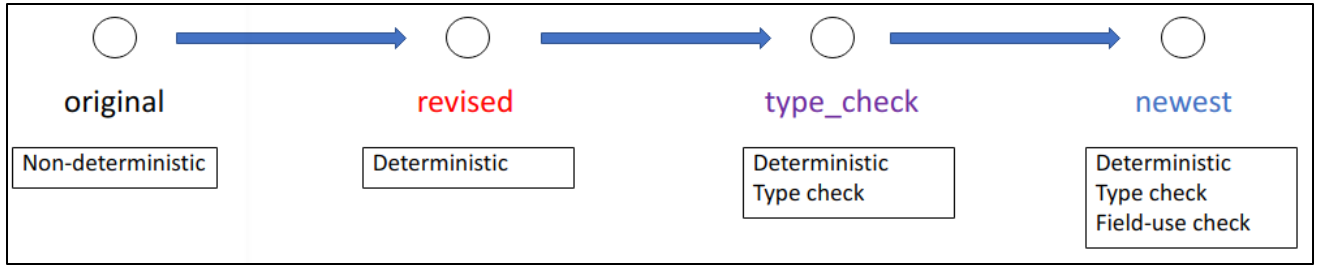
Pta has detected 8 cases.



Evaluation – Q3

- Q3: How effective is field-use analysis?

- Check diff between type_check(with pta) and newest(with pta):

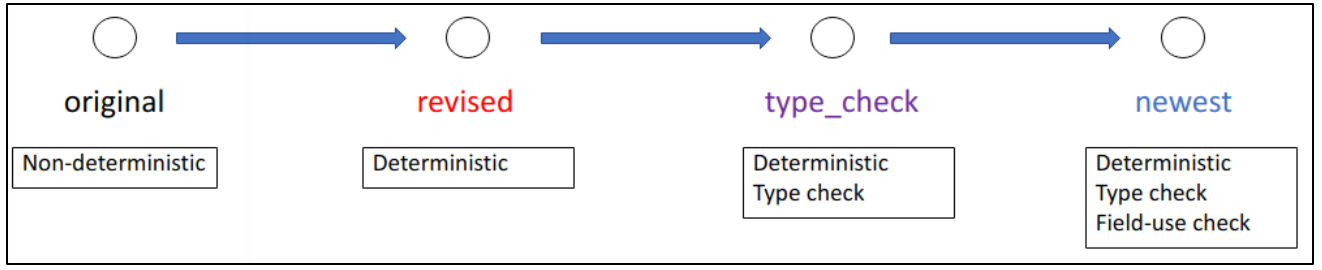


Evaluation – Q3

- Q3: How effective is field-use analysis?

- Check diff between type_check(with pta) and newest(with pta):
  - newest removes 64 sink taints, where
    - ✓ 11 of them taints the field that is never used and
    - ✓ 53 of them taints the field that is unknown to be used in sink

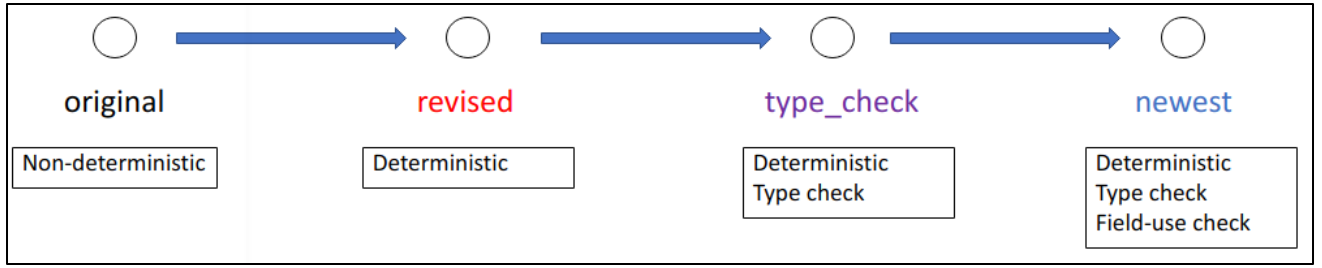
Field-use check can remove some false-positive cases.



Evaluation – Q4

- Q4: What is the performance cost of new cflow?

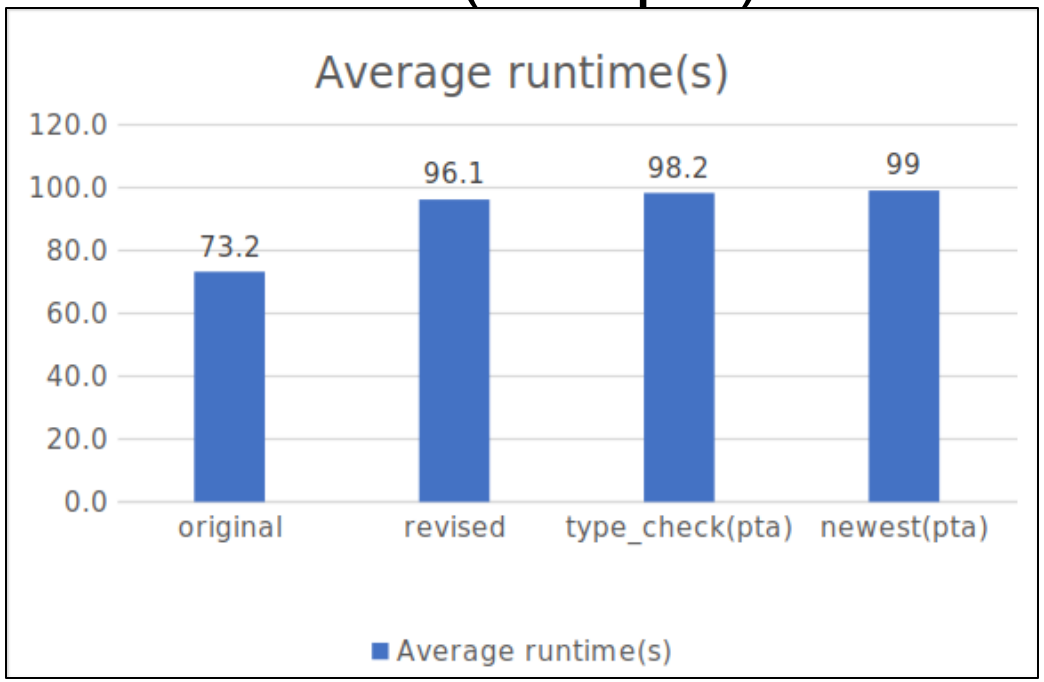
- I compare the avg runtime of original, revised, type_check(with pta) and newest(with pta).



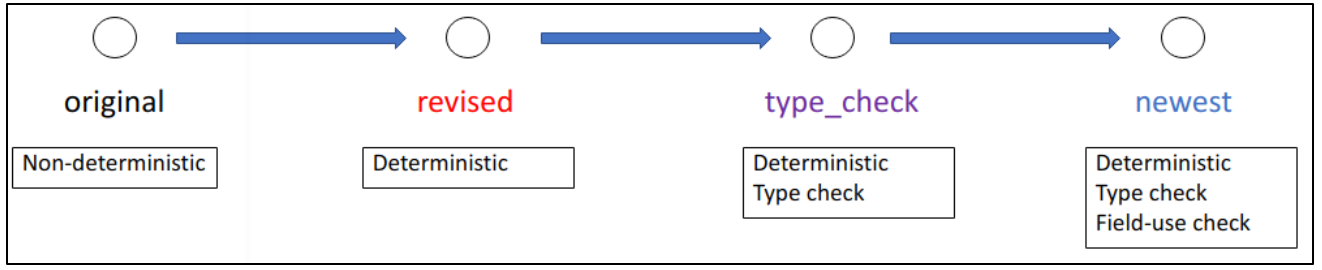
Evaluation – Q4

- Q4: What is the performance cost of new cflow?

- I compare the avg runtime of original, revised, type_check(with pta) and newest(with pta).



Pta and field-use analysis don't have a big overhead.



Content

- Review
- Problem
- Solution
- Evaluation
- Conclusion

Conclusion

- In my work,
  - I add flow information to let cflow have deterministic output.
  - I add points-to analysis and field-use analysis to reduce false-positives.

- However,
  - The effect of type check is limited.
  - Cflow should be tested on more applications.

Thanks

Nov xth, 2021