Merge-Replay: Efficient IFDS-Based Taint Analysis by Consolidating Equivalent Value Flows

Yujiang Gui*, Dongjie He*† and Jingling Xue†

School of Computer Science and Engineering
University of New South Wales

Presenter: Yujiang Gui

38th ASE, September 2023

∗, † The first two authors contributed equally and are listed in alphabetical order by their last names, while the last two authors share corresponding authorship.

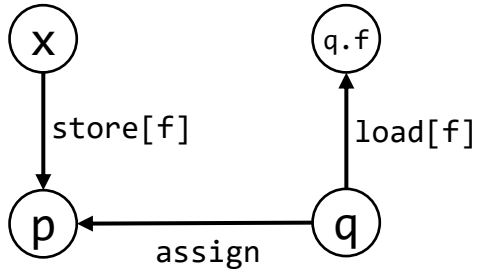
Static Taint Analysis

- Exposes potential sensitive data leaks ahead of time
  - Private messages
  - Location information
  - Financial details
  - …

Static Taint Analysis

- Exposes potential sensitive data leaks ahead of time
  - Private messages
  - Location information
  - Financial details
  - …
- Tracks the flows of tainted data to observe if they can reach sink

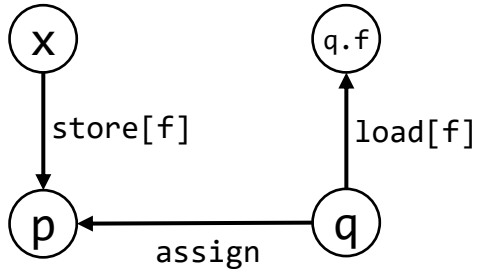
```
x = getPassword(); // source
p = q;
...
p.f = x;
sendMessage(q.f);  // sink
```



Static Taint Analysis

- Exposes potential sensitive data leaks ahead of time
  - Private messages
  - Location information
  - Financial details
  - …
- Tracks the flows of tainted data to observe if they can reach sink

```
x = getPassword(); // source
p = q;
...
p.f = x;
sendMessage(q.f);  // sink
```



- An active research area
  - Security analysis for web apps (ASE'14, ISSTA'23)
  - Taint analysis for Android apps (PLDI'14, ISSTA'15, OOPSLA'18, ICSE'20)
  - Memory leak detection (CGO'21, ICSE'21, ISSTA'23)
  - …

IFDS-Based Taint Analysis

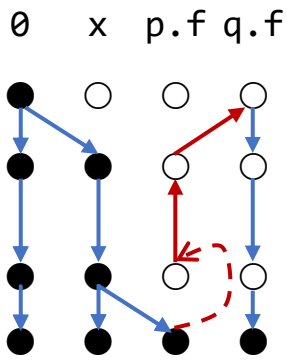
- Utilizes the IFDS framework*
  - Solves a class of interprocedural distributive analyses
  - Transforms an analysis into a graph-reachability problem
  - Context- and flow-sensitive

*Reps, Thomas, Susan Horwitz, and Mooly Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability.”, POPL’95.

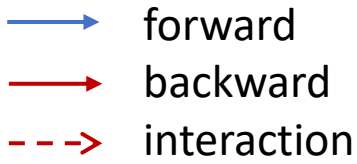
IFDS-Based Taint Analysis

- Utilizes the IFDS framework*
  - Solves a class of interprocedural distributive analyses
  - Transforms an analysis into a graph-reachability problem
  - Context- and flow-sensitive

- Employs two mutually iterative passes
  - Identify taints forwards
  - Detect aliases backwards



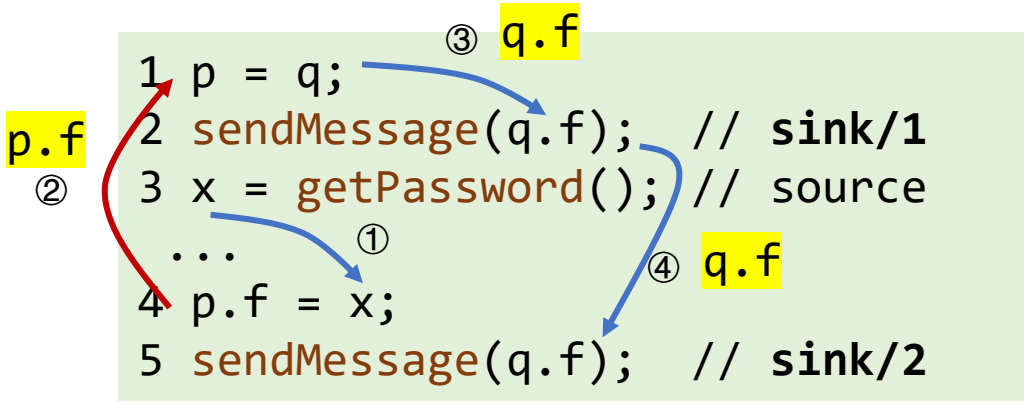

```
x = getPassword(); // source
p = q;
...
p.f = x;
sendMessage(q.f);  // sink
```



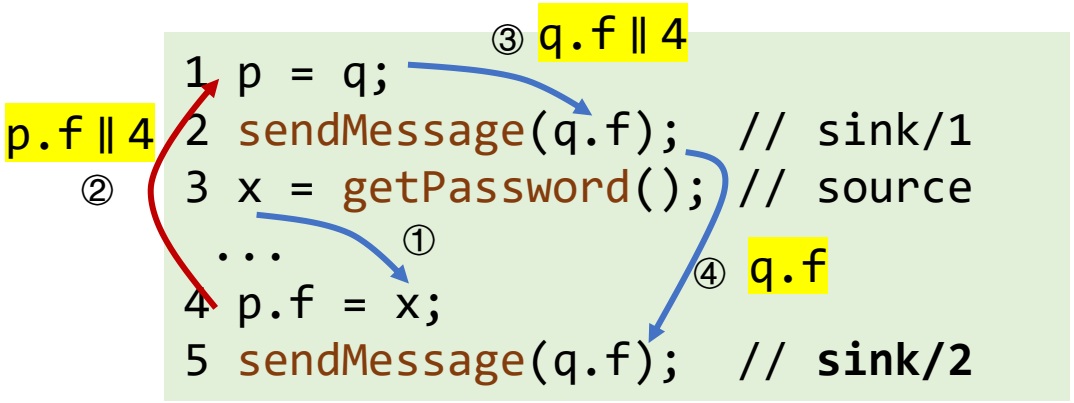
*Reps, Thomas, Susan Horwitz, and Mooly Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability.”, POPL’95.

Maintaining Flow Sensitivity with Activation Statements

- Loss of flow sensitivity during the interactions of solvers



- FlowDroid*
  - Recovers flow sensitivity with activation statements



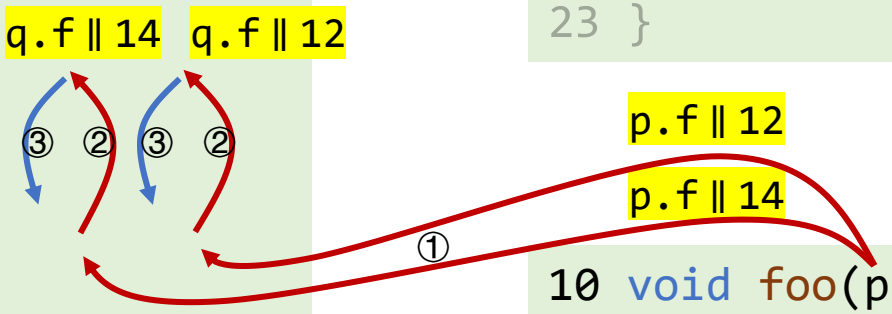
*Arzt, Steven, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. "FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps.", PLDI'14.

Limitation 1: Redundant Propagation

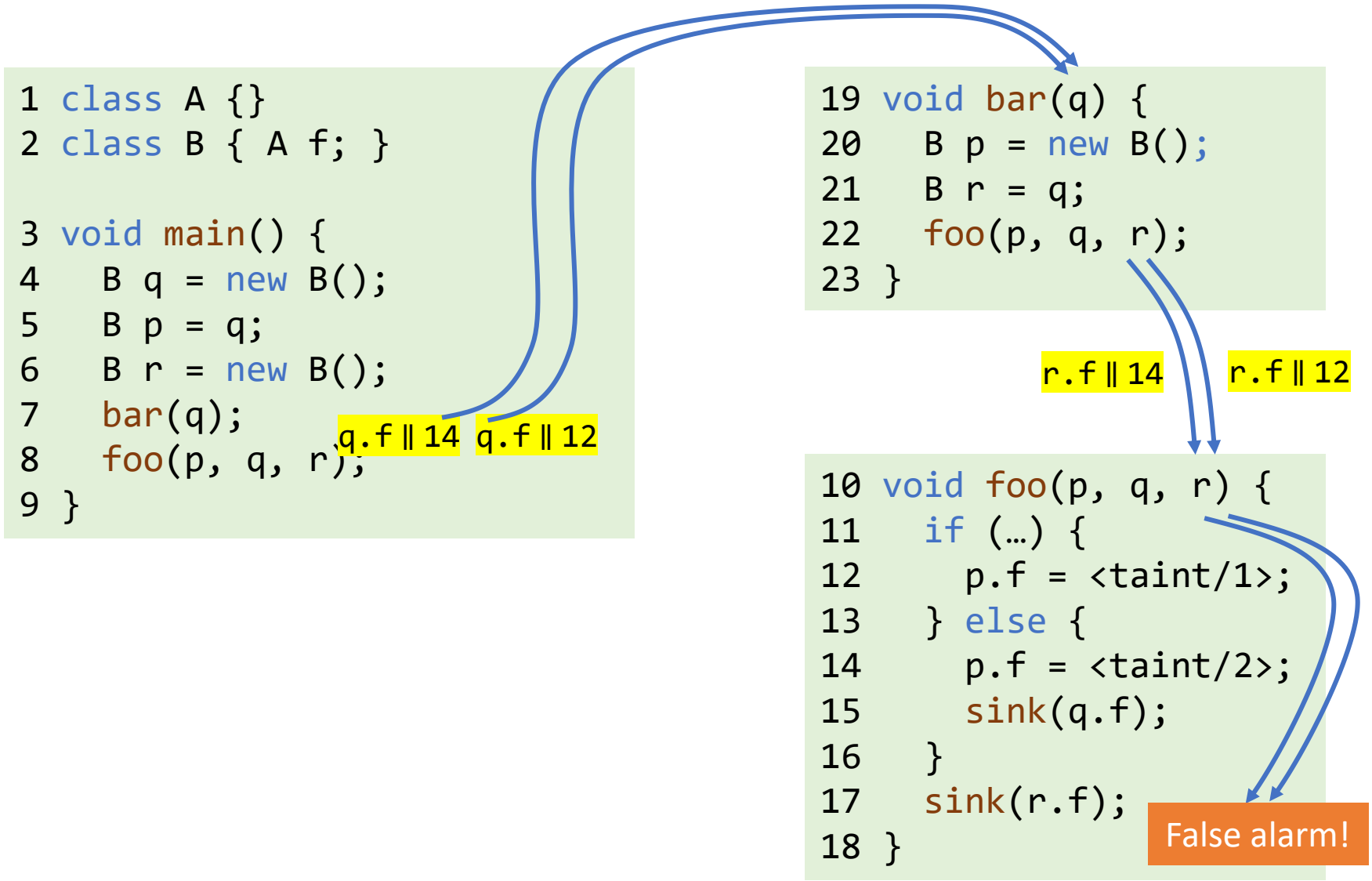
```
1 class A {}
2 class B { A f; }

3 void main() {
4   B q = new B();
5   B p = q;
6   B r = new B();
7   bar(q);
8   foo(p, q, r);
9 }
```

```
19 void bar(q) {
20   B p = new B();
21   B r = q;
22   foo(p, q, r);
23 }
```

```
10 void foo(p, q, r) {
11   if (…) {
12     p.f = <taint/1>;
13   } else {
14     p.f = <taint/2>;
15     sink(q.f);
16   }
17   sink(r.f);
18 }
```



Limitation 2: Overlooking Context

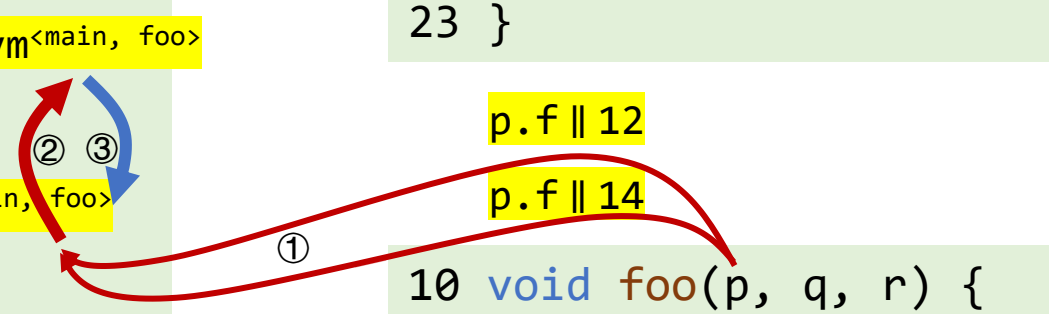


The Merge-Replay Strategy

```
1 class A {}
2 class B { A f; }

3 void main() {
4   B q = new B();
5   B p = q;
6   B r = new B();
7   bar(q);
8   foo(p, q, r);
9 }
```

```
19 void bar(q) {
20   B p = new B();
21   B r = q;
22   foo(p, q, r);
23 }
```

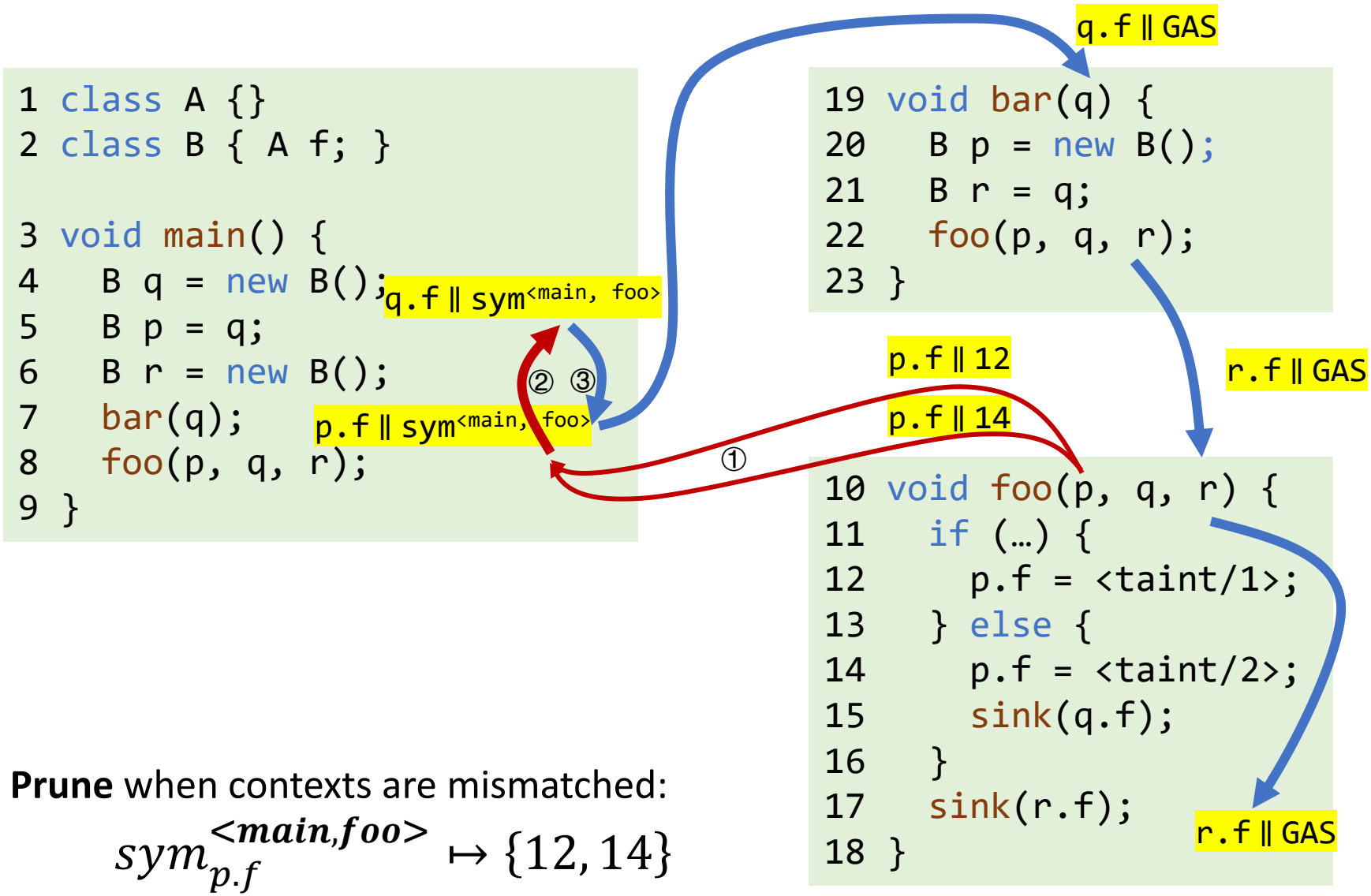
```
10 void foo(p, q, r) {
11   if (…) {
12     p.f = <taint/1>;
13   } else {
14     p.f = <taint/2>;
15     sink(q.f);
16   }
17   sink(r.f);
18 }
```



Merge with a symbolic activation statement:

$$\boldsymbol{sym}_{\boldsymbol{p.f}}^{<main,foo>} \mapsto \{12, 14\}$$

The Merge-Replay Strategy



Prune when contexts are mismatched:

$$sym_{p.f}^{<main,foo>} \mapsto \{12, 14\}$$

The Merge-Replay Strategy

```
1 class A {}
2 class B { A f; }

3 void main() {
4   B q = new B();
5   B p = q;
6   B r = new B();
7   bar(q);
8   foo(p, q, r);
9 }
```

①

q.f ‖ sym<main, foo>

②

```
19 void bar(q) {
20   B p = new B();
21   B r = q;
22   foo(p, q, r);
23 }
```

```
10 void foo(p, q, r) {
11   if (…) { q.f ‖ 12  q.f ‖ 14
12     p.f = <taint/1>;
13   } else {
14     p.f = <taint/2>;
15     sink(q.f);
16   }
17   sink(r.f);
18 }
```

Replay using concrete activation statements:

$$sym_{p.f}^{<main,foo>} \mapsto \{12, 14\}$$

Summary

- Reduce redundant propagation
- Enhance precision
- Boost overall performance
- Conceptually simple

The MergeDroid Algorithm

```
function ForwardAnalysis()
11a   for d3 such that ⟨n, d2⟩ → ⟨s_m', d3⟩ ∈ E#_FW do
11b     for d'3 ∈ Concretize(⟨m, d1⟩ → ⟨n, d2⟩, m', d3) do
12a       Inject(⟨m, d1⟩ → ⟨n, d2⟩, ⟨m', d'3⟩, E#_BW, PathEdge_BW, S_BW, W_BW)
13a       Prop(⟨m', d'3⟩ → ⟨s'_m, d'3⟩, W_FW, PathEdge_FW)
14a       for ⟨m', d'3⟩ → ⟨e_m', d4⟩ ∈ S_FW ∧ ⟨e_m', d4⟩ → ⟨r, d5⟩ ∈ E#_FW do
14b         d'5 = AttachActivationStmt(d5, d2)
15a         Prop(⟨m, d1⟩ → ⟨r, d'5⟩, W_FW, PathEdge_FW)
    ...
22a   for ⟨m'', d3⟩ → ⟨c, d4⟩ ∈ PathEdge_FW ∧ ⟨c, d4⟩ → ⟨s_m, d1⟩ ∈ E#_FW
        ∧⟨n, d2⟩ → ⟨r, d5⟩ ∈ E#_FW do
22b     d'5 = AttachActivationStmt(d5, d4)
23a     Prop(⟨m'', d3⟩ → ⟨r, d'5⟩, W_FW, PathEdge_FW)
function BackwardAnalysis()
38a   for d3 such that ⟨n, d2⟩ → ⟨e_m', d3⟩ ∈ E#_BW do
38b     d'3 = DataAbstraction(d3) ‖ GAS
39a     Inject(⟨m, d1⟩ → ⟨n, d2⟩, ⟨m', d'3⟩, E#_FW, PathEdge_FW, S_FW, W_FW)
40a     Prop(⟨m', d'3⟩ → ⟨e_m', d'3⟩, W_BW, PathEdge_BW)
41a     for ⟨m', d'3⟩ → ⟨s_m', d4⟩ ∈ S_BW ∧ ⟨s_m', d4⟩ → ⟨r, d5⟩ ∈ E#_BW do
41b       d'5 = OnReturnFlow(d5, d2, m, m')
42a       Prop(⟨m, d1⟩ → ⟨r, d'5⟩, W_BW, PathEdge_BW)
    ...
49a   for ⟨m'', d3⟩ → ⟨c, d4⟩ ∈ PathEdge_BW ∧ ⟨c, d4⟩ → ⟨e_m, d1⟩ ∈ E#_BW
        ∧⟨n, d2⟩ → ⟨r, d5⟩ ∈ E#_BW do
49b     d'5 = OnReturnFlow(d5, d4, m'', m)
50a     Prop(⟨m'', d3⟩ → ⟨r, d'5⟩, W_BW, PathEdge_BW)
61  function OnActivationStmtAdded(sym, as)
62    ⟨m, m'⟩ = Context(sym)    // m is caller, m' is callee
63    for ⟨⟨m, d1⟩ → ⟨c, d2⟩, abs⟩ ∈ SymbolIncoming(sym) do
64      d3 = abs ‖ as
65      Prop(⟨m', d3⟩ → ⟨s_m', d3⟩, W_FW, PathEdge_FW)
66      Let r be the return node of c
67      for ⟨m', d3⟩ → ⟨e_m', d4⟩ ∈ S_FW ∧ ⟨e_m', d4⟩ → ⟨r, d5⟩ ∈ E#_FW do
68        Prop(⟨m, d1⟩ → ⟨r, d5⟩, W_FW, PathEdge_FW)
```

```
69  SymbolIncoming = Symb2Reps = {}
70  function Symbolize(d_ret, caller, callee)
71    as = ActivationStmt(d_ret)
72    abs = DataAbstraction(d_ret)
73    sym = ⟨abs, caller, callee⟩    // Symbolic activation stmt
74    if as ∉ Symb2Reps(sym) then
75      Symb2Reps(sym) ∋ as
76      OnActivationStmtAdded(sym, as)
77    return abs ‖ sym
78  function Concretize(⟨caller, d1⟩ → ⟨c, d2⟩, callee, d3)
79    if d3 is active then return {d3}
80    as = ActivationStmt(d3)
81    if as is a concrete statement or as = GAS then return {d3}
82    assert as is a symbolic activation statement
83    sym = as                                  // Rename variable
84    reps = {}                                 // Set of represented facts
85    abs = DataAbstraction(d3)
86    if Context(sym) = ⟨caller, callee⟩ then
87      SymbolIncoming(sym) ∋ ⟨⟨caller, d1⟩ → ⟨c, d2⟩, abs⟩
88      for v ∈ Symb2Reps(sym) do
89        reps ∋ abs ‖ v
90    else
91      reps ∋ abs ‖ GAS
92    return reps
93  function AttachActivationStmt(d_ret, d_call)
94    u = ActivationStmt(d_ret)
95    v = ActivationStmt(d_call)
96    if u = GAS then
97      return DataAbstraction(d_ret) ‖ v
98    return d_ret
99  function OnReturnFlow(d_ret, d_call, caller, callee)
100   u = ActivationStmt(d_ret)
101   if u = GAS then
102     return AttachActivationStmt(d_ret, d_call)
103   return Symbolize(d_ret, caller, callee)
```

Implementation

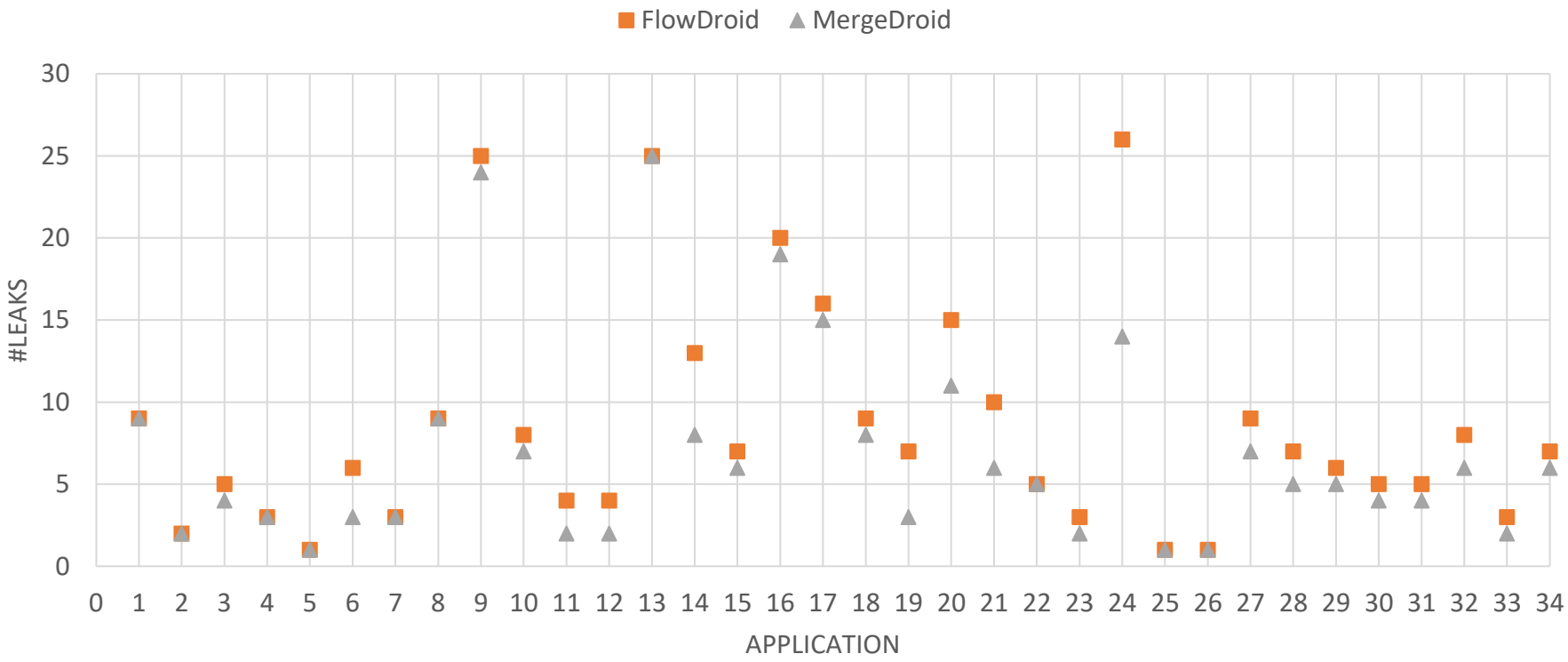
- Implemented on top of **FlowDroid** (PLDI'14)
- In about **400 lines** of Java code
- Available at https://www.cse.unsw.edu.au/~corg/MergeDroid

Evaluation

- Benchmark (40 apps)
  - From previous papers (ASE'19 and CGO'21)
  - From F-Droid
- Analysis budget
  - 3 hours and 256GB per app
- Metrics
  - Precision
  - Analysis time
  - Memory usage

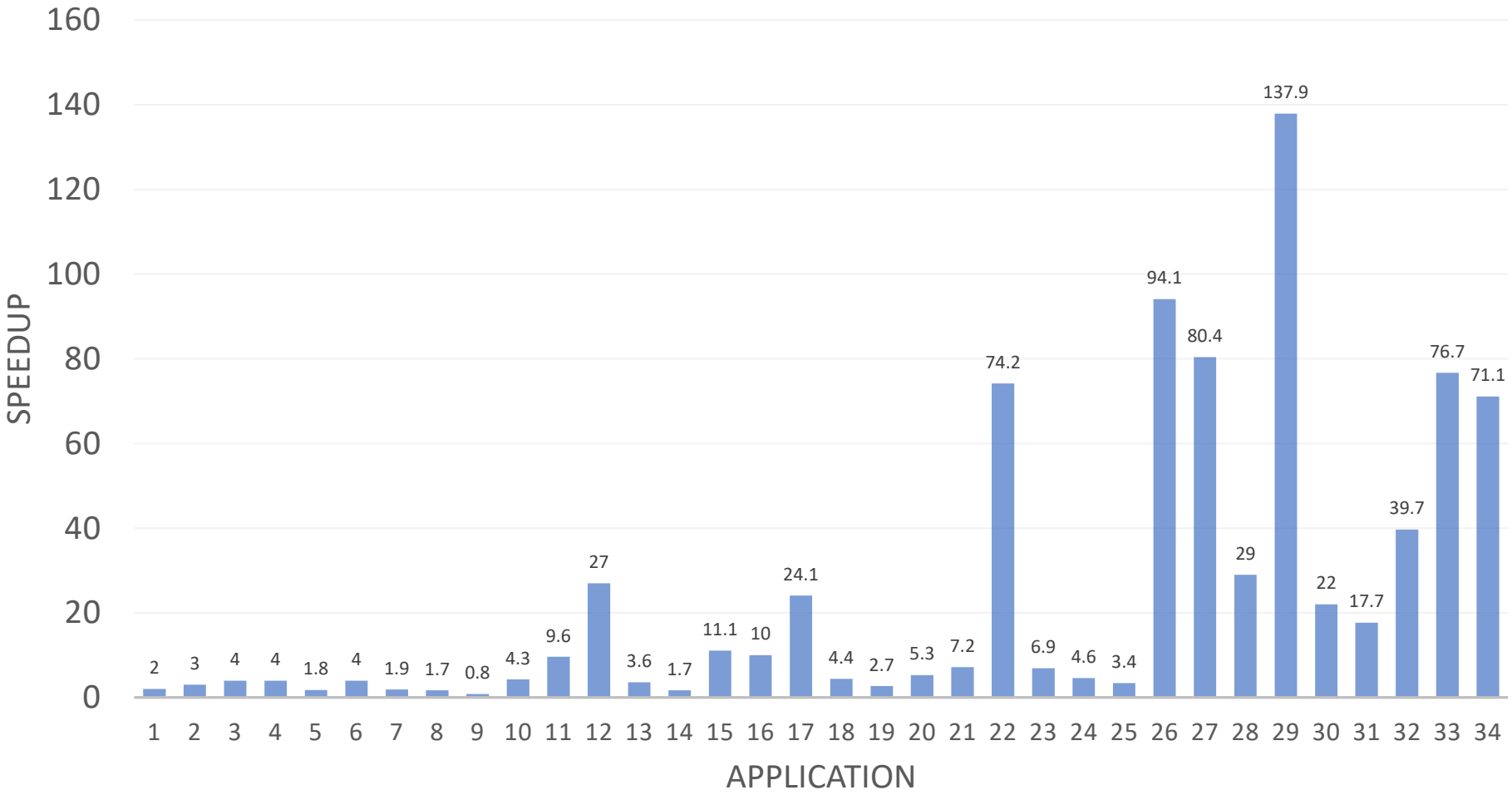
RQ1: Precision

- Validated correctness using
  - DroidBench
  - TaintBench



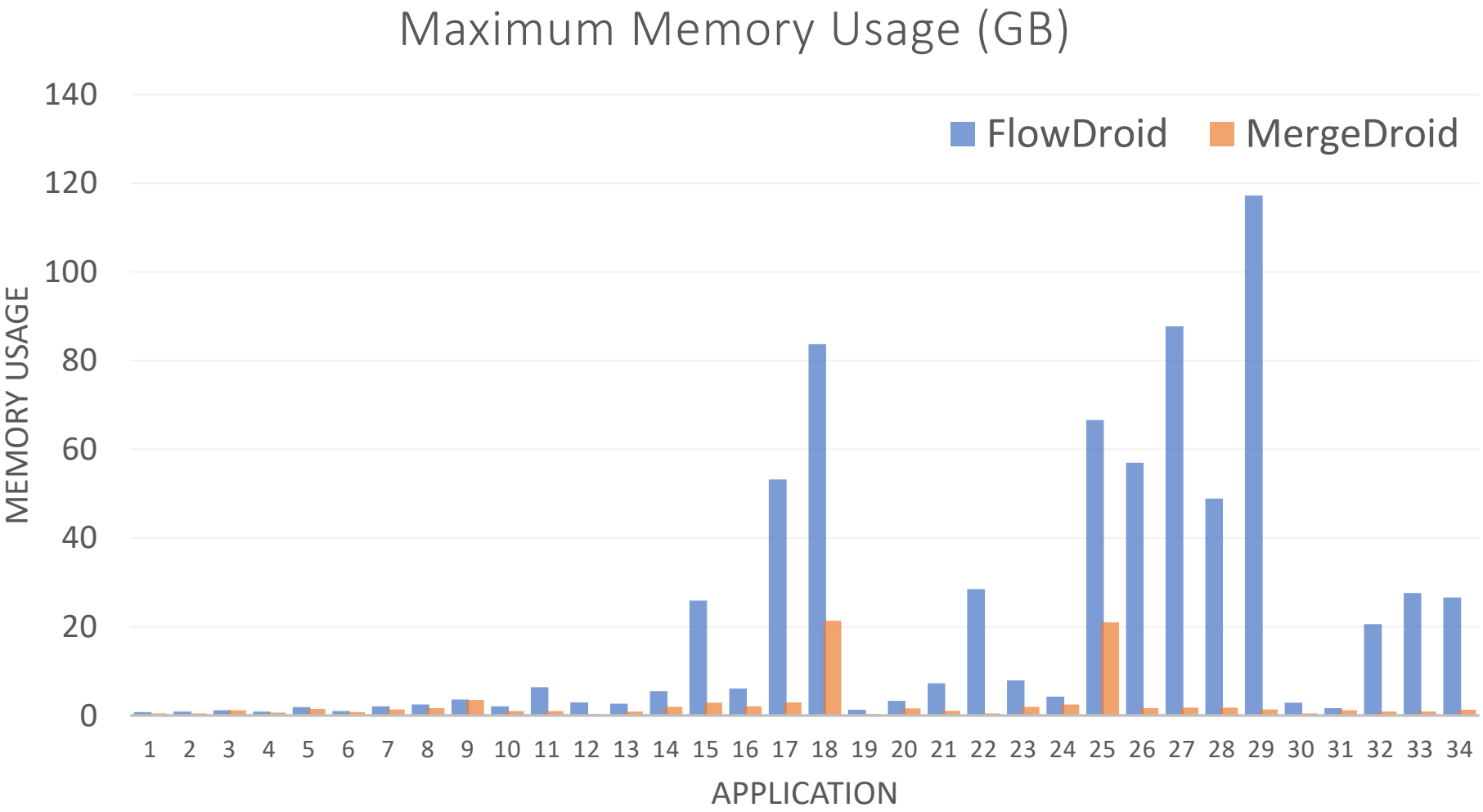
MergeDroid reduces false positives by decreasing reported leak warnings by FlowDroid by 19.2% on average.

RQ2: Speedups



MergeDroid improves the efficiency of FlowDroid, and scales 6 more apps. The speedups range from 0.8× to 137.9× with an average of 9.0×.

RQ3: Memory Requirements



MergeDroid uses less memory than FlowDroid for all apps analyzed, the ratio range from 1.0× to 83.6× with an average of 5.2×.

Q & A

Thank you!