Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis

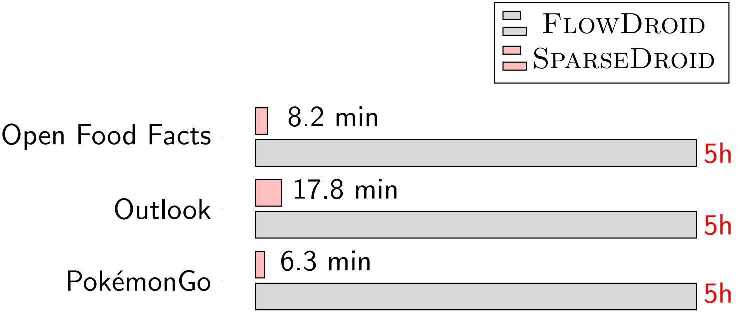
Dongjie He[1,2,3], Haofeng Li[2,3], Lei Wang[2,3], Haining Meng[2,3], Hengjie Zheng[2,3], Jie Liu[1], Shuangwei Hu[4], Lian Li*[2,3] and Jingling Xue* [1]

[1]UNSW Sydney, Australia
[2]SKL of Computer Architecture, ICT, CAS, China
[3]University of Chinese Academy of Sciences, China
[4]Vivo AI Lab, China

*Corresponding author

Our Work: SparseDroid

- Fast

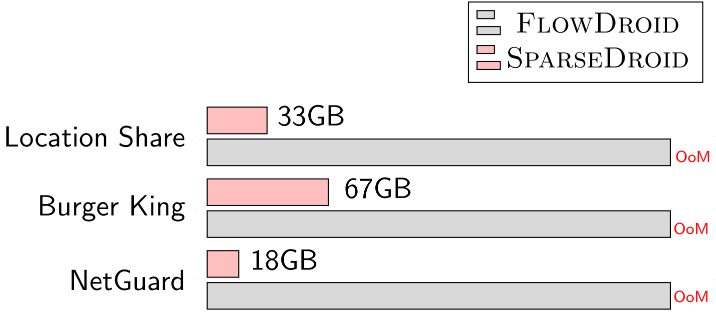


Our Work: SparseDroid

- Memory-efficient

NetGuard

No-root firewall



- OoM: Out of Memory

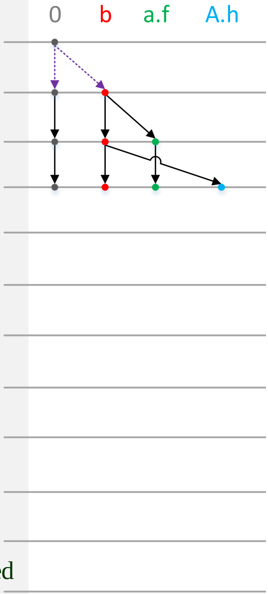
How have we achieved this ?

Non-Sparse IFDS-based Taint Analysis (FlowDroid)

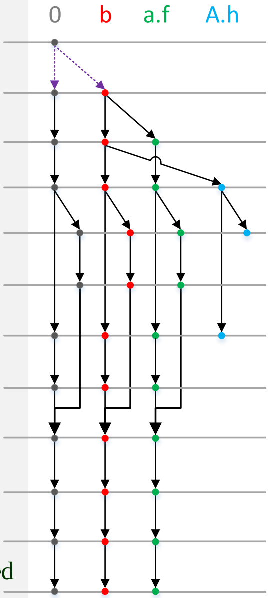
```
void foo(a) {
    b = source();
    a.f = b;
    A.h = b;
    if (···) {
        A.h = ··· ; // kill
    } else {
        A.h = ··· ; // kill
    }
    ··· // irrelevant code
    sink(a.f); // tainted
    sink(A.h); // untainted
}
```

```
void foo(a) {
    b = source();
    a.f = b;
    A.h = b;
    if (···) {
        A.h = ··· ; // kill
    } else {
        A.h = ··· ; // kill
    }
    ··· // irrelevant code
    sink(a.f); // tainted
    sink(A.h); // untainted
}
```

```
void foo(a) {
    b = source();
    a.f = b;
    A.h = b;
    if (···) {
        A.h = ··· ; // kill
    } else {
        A.h = ··· ; // kill
    }
    ··· // irrelevant code
    sink(a.f); // tainted
    sink(A.h); // untainted
}
```

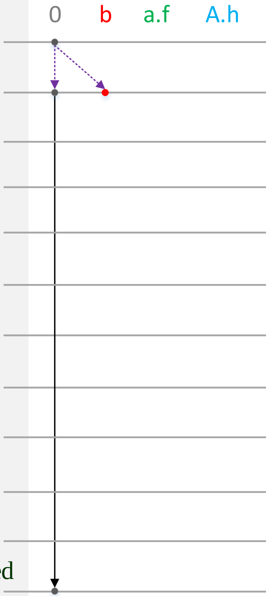


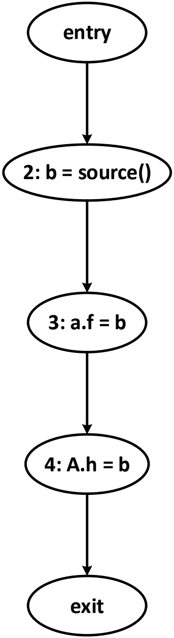
```
void foo(a) {
    b = source();
    a.f = b;
    A.h = b;
    if  (···) {
        A.h = ··· ; // kill
    } else {
        A.h = ··· ; // kill
    }
    ··· // irrelevant code
    sink(a.f); // tainted
    sink(A.h); // untainted
}
```



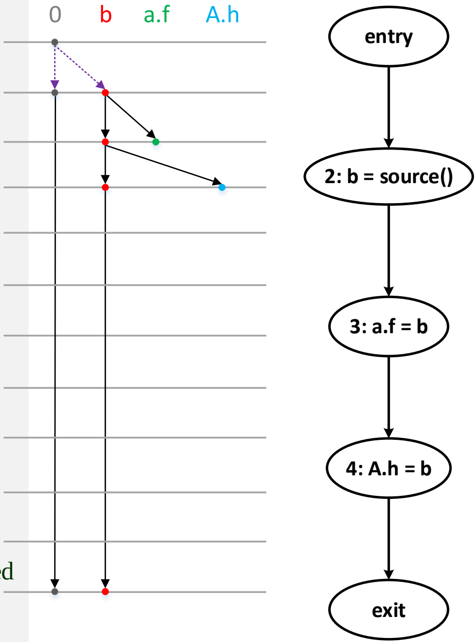
Sparse IFDS-based Taint Analysis (SparseDroid)

```
void foo(a) {
    b = source();
    a.f = b;
    A.h = b;
    if  (···) {
        A.h = ··· ; // kill
    } else {
        A.h = ··· ; // kill
    }
    ··· // irrelevant code
    sink(a.f); // tainted
    sink(A.h); // untainted
}
```

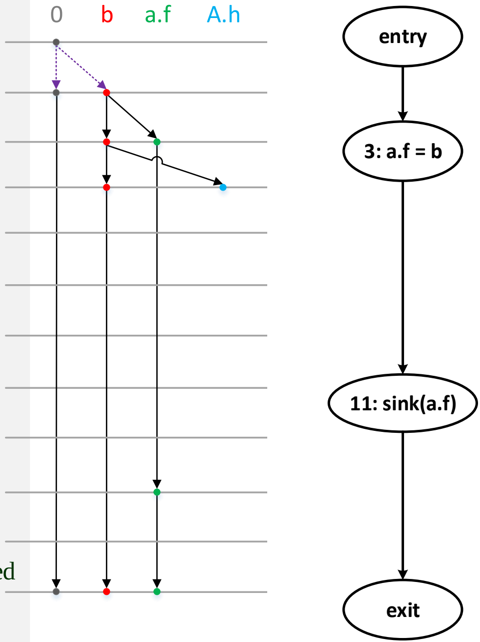




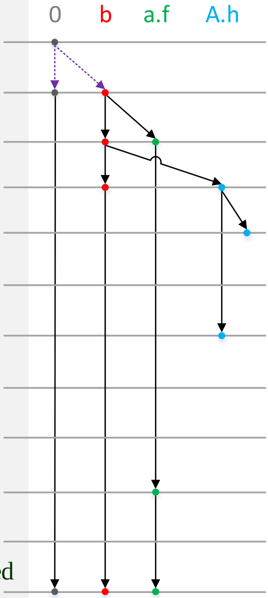
```
1  void foo(a) {
2      b = source();
3      a.f = b;
4      A.h = b;
5      if (···) {
6          A.h = ··· ; // kill
7      } else {
8          A.h = ··· ; // kill
9      }
10     ··· // irrelevant code
11     sink(a.f); // tainted
12     sink(A.h); // untainted
13 }
```

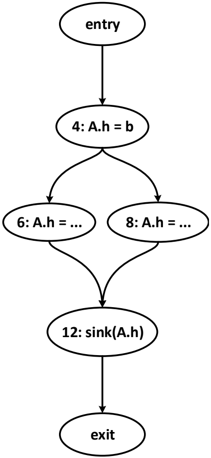


```
1  void foo(a) {
2      b = source();
3      a.f = b;
4      A.h = b;
5      if (···) {
6          A.h = ··· ; // kill
7      } else {
8          A.h = ··· ; // kill
9      }
10     ··· // irrelevant code
11     sink(a.f); // tainted
12     sink(A.h); // untainted
13 }
```

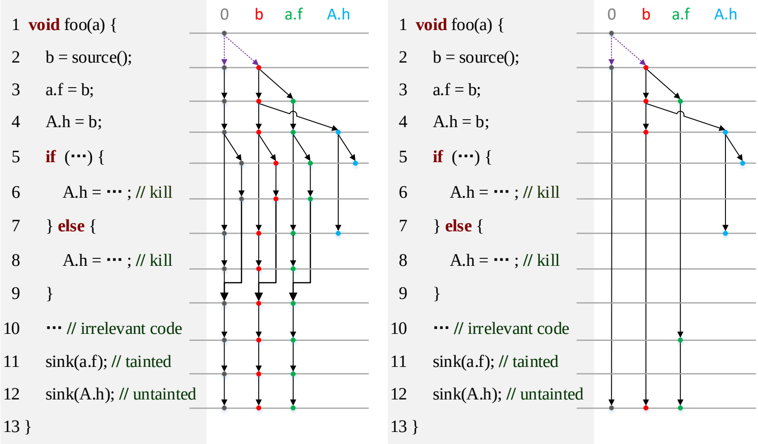


```
void foo(a) {
    b = source();
    a.f = b;
    A.h = b;
    if (···) {
        A.h = ··· ; // kill
    } else {
        A.h = ··· ; // kill
    }
    ··· // irrelevant code
    sink(a.f); // tainted
    sink(A.h); // untainted
}
```





Property 1: Sparsity

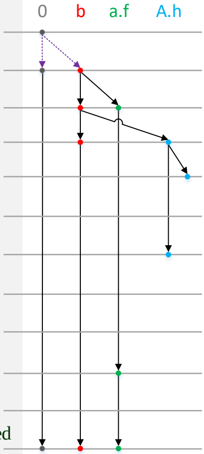


- Non-Sparse IFDS propagates facts to next statements
- Sparse IFDS sends facts to their next use statements

Property 1: Sparsity

- **Key Observation:** *fact-specific identity functions*

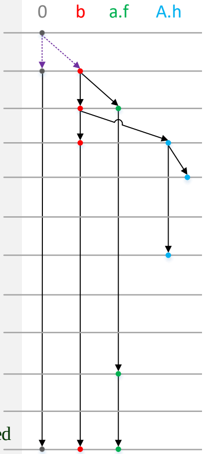
```
void foo(a) {
    b = source();
    a.f = b;
    A.h = b;
    if (···) {
        A.h = ··· ; // kill
    } else {
        A.h = ··· ; // kill
    }
    ··· // irrelevant code
    sink(a.f); // tainted
    sink(A.h); // untainted
}
```



- For fact *d*, skip edges of *d*-specific identity function

Property 2: Multi-threading

```
void foo(a) {
    b = source();
    a.f = b;
    A.h = b;
    if (···) {
        A.h = ··· ; // kill
    } else {
        A.h = ··· ; // kill
    }
    ··· // irrelevant code
    sink(a.f); // tainted
    sink(A.h); // untainted
}
```



- Flow functions are distributive
    - independent propagation
- Orthogonal to multi-threading parallelization

Property 3: On-demand SCFG construction

- build sparse control flow graph (SCFG) for each fact on-demand
- cache already built SCFG for reuse

Property 3: On-demand SCFG construction

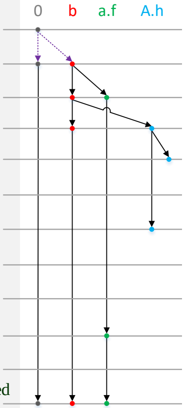
- build sparse control flow graph (SCFG) for each fact on-demand
- cache already built SCFG for reuse

- Why not build SCFG in a pre-analysis?
  - hard to predict potential data flow facts
  - over-approximate facts incur unnecessary over-head

Property 4: Precision and efficiency

- Maintain precision

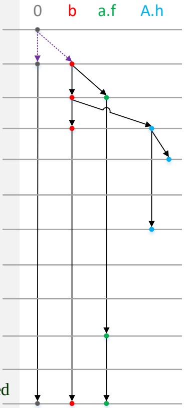
```
void foo(a) {
    b = source();
    a.f = b;
    A.h = b;
    if (···) {
        A.h = ··· ; // kill
    } else {
        A.h = ··· ; // kill
    }
    ··· // irrelevant code
    sink(a.f); // tainted
    sink(A.h); // untainted
}
```



Property 4: Precision and efficiency

- Maintain precision

```
void foo(a) {
    b = source();
    a.f = b;
    A.h = b;
    if (···) {
        A.h = ··· ; // kill
    } else {
        A.h = ··· ; // kill
    }
    ··· // irrelevant code
    sink(a.f); // tainted
    sink(A.h); // untainted
}
```



- Boost performance
  - save fewer facts
  - handle fewer flow functions

Sparse IFDS Algorithm

Function `ForwardTabulateSLRPs()`

while $WorkList \neq \emptyset$ **do**

...

switch n **do**

case $n \in callsites(p)$ **do**

...

for d_3 *s.t.* $\langle n, d_2 \rangle \rightarrow \langle retSite(n), d_3 \rangle \in (E^\# \cup \mathcal{S})$ **do**

if $\mathcal{G}^\#_{p,d_3}$ *is not constructed yet, i.e., not in the SCFG cache* **then**

Build $\mathcal{G}^\#_{p,d_3} = (\mathcal{N}^\#_{p,d}, \mathcal{E}^\#_{p,d})$ according to (2)

for $\langle n, d_2 \rangle \rightarrow \langle m', d_3 \rangle \in \mathcal{E}^\#_{p,d_3}$ **do**

`Prop(`$\langle s_p, d_1 \rangle \rightarrow \langle m', d_3 \rangle$`)`

case $n = e_p$ **do**

...

case $n \in (N_p - callsites(p) - \{e_p\})$ **do**

for $\langle \_, d_3 \rangle$ *s.t.* $\langle n, d_2 \rangle \rightarrow \langle \_, d_3 \rangle \in E^\#$ **do**

if $\mathcal{G}^\#_{p,d_3}$ *is not constructed yet, i.e., not in the SCFG cache* **then**

Build $\mathcal{G}^\#_{p,d_3} = (\mathcal{N}^\#_{p,d}, \mathcal{E}^\#_{p,d})$ according to (2)

for $\langle n, d_2 \rangle \rightarrow \langle m', d_3 \rangle \in \mathcal{E}^\#_{p,d_3}$ **do**

`Prop(`$\langle s_p, d_1 \rangle \rightarrow \langle m', d_3 \rangle$`)`

SCFG Construction

- d-specific identity function

$$\begin{aligned} \forall\ X \in 2^{D} \quad &: \quad d \in X \implies d \in f(X) \\ \forall\ X \in 2^{D\setminus\{d\}} \quad &: \quad f(X)\setminus\{d\} = f(X \cup \{d\})\setminus\{d\} \end{aligned} \tag{1}$$

- SCFG $\mathcal{G}_p^d = (\mathcal{N}_{p,d}^{\#}, \mathcal{E}_{p,d}^{\#})$

$$\begin{aligned} \mathcal{E}_{p,d}^{\#} &= \{(m,d) \to (n,d') \in E_p^{\#}) \mid M(m,n) \not\equiv M(m,n)^d\} \\ &\cup \{(m,d) \to (n,d) \mid P_p^d(m,n) \text{ is sparsifiable}\} \\ \mathcal{N}_{p,d}^{\#} &= \bigcup\nolimits_{(m,d)\to(n,d')\in\mathcal{E}_{p,d}^{\#}} \{(m,d)\} \cup \{(n,d')\} \end{aligned} \tag{2}$$

Flow Functions for Taint Analysis

$$\frac{a = source() \quad \{\mathbf{0}\}}{\{\mathbf{0}, a.*\}} \qquad \frac{a = source() \quad \{v.f\} \quad v \neq a}{\{v.\bar{f}\}} \quad \text{[SOURCE]} \qquad \frac{a = ... \quad \{v.f\} \quad v = a}{\{\}} \quad \text{[KILL]}$$

$$\frac{a = new\ T() \quad \{v.\bar{f}\} \quad v \neq a}{\{v.\bar{f}\}} \quad \text{[NEW]} \qquad \frac{a = b \quad \{v.\bar{f}\} \quad v = b}{\{v.\bar{f}, a.\bar{f}\}} \qquad \frac{a = b \quad \{v.\bar{f}\} \quad v \notin \{a, b\}}{\{v.\bar{f}\}} \quad \text{[ASSIGN]}$$

$$\frac{a_2 = \phi(a_0, a_1) \quad \{v.\bar{f}\} \quad v \in \{a_0, a_1\}}{\{v.\bar{f}, a_2.\bar{f}\}} \qquad \frac{a_2 = \phi(a_0, a_1) \quad \{v.\bar{f}\} \quad v \notin \{a_0, a_1, a_2\}}{\{v.\bar{f}\}} \quad \text{[PHI]}$$

$$\frac{a = \xi.f' \ (\xi \in \{b, T\}) \quad \{v.\bar{f}\} \quad v = \xi \wedge \mathbf{car}(\bar{f}) = f'}{\{v.\bar{f}, a.\mathbf{cdr}(\bar{f})\}} \qquad \frac{a = \xi.f' \ (\xi \in \{b, T\}) \quad \{v.\bar{f}\} \quad v \neq a \wedge (v \neq \xi \vee \mathbf{car}(\bar{f}) \neq f')}{\{v.\bar{f}\}} \quad \text{[LOAD]}$$

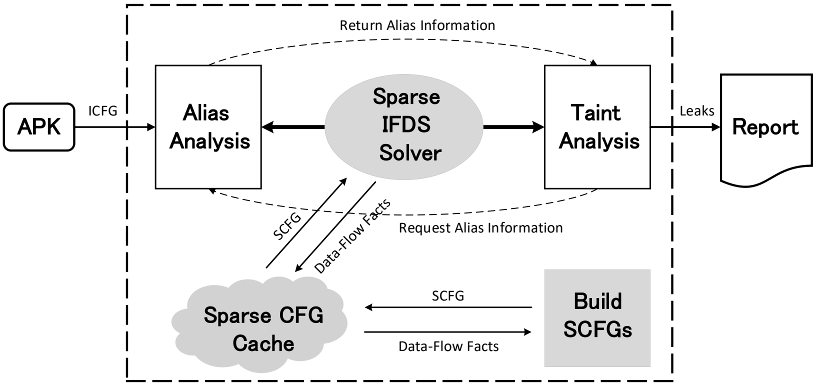
$$\frac{f' = b \ (\xi \in \{a, T\}) \quad \{v.\bar{f}\} \quad v = b}{\{v.\bar{f}, \xi.f'.\bar{f}\}} \qquad \frac{\xi.f' = b \ (\xi \in \{a, T\}) \quad \{v.\bar{f}\} \quad v = \xi \wedge \mathbf{car}(\bar{f}) = f'}{\{\}} \qquad \frac{\xi.f' = b \ (\xi \in \{a, T\}) \quad \{v.\bar{f}\} \quad v \neq b \wedge (v \neq \xi \vee \mathbf{car}(\bar{f}) \neq f')}{\{v.\bar{f}\}} \quad \text{[STORE]}$$

$$\frac{r = foo(\bar{a}) \quad \{v.\bar{f}\} \quad v = a_i \quad a_i \in \bar{a} \quad p_i \text{ is } a_i\text{'s corresponding formal parameter in } foo}{\{p_i.\bar{f}\}} \qquad \frac{r = foo(\bar{a}) \quad \{v.\bar{f}\} \quad v \notin \bar{a}}{\{\}} \qquad \frac{r = foo(\bar{a}) \quad \{T.\bar{f}\}}{\{T.\bar{f}\}} \quad \text{[CALL]}$$

$$\frac{r = foo(\bar{a}) \quad \{v.\bar{f}\} \quad v \in \bar{a} \vee v = T}{\{\}} \qquad \frac{r = foo(\bar{a}) \quad \{v.\bar{f}\} \quad v \notin \bar{a} \cup \{r\} \wedge v \neq T}{\{v.\bar{f}\}} \quad \text{[CALL-TO-RETURN]}$$

$$\frac{ret_{foo}\ r \quad \{v.\bar{f}\} \quad v = p_i \quad p_i \text{ is } foo\text{'s formal parameter} \quad a_i \text{ is } p_i\text{'s corresponding actual argument}}{\{a_i.\bar{f}\}} \qquad \frac{ret_{foo}\ r_0 \quad \{v.\bar{f}\} \quad v = r_0 \quad r_1 = foo(\bar{a}) \in callers(foo)}{\{r_1.\bar{f}\}} \qquad \frac{ret_{foo}\ r \quad \{T.\bar{f}\}}{\{T.\bar{f}\}} \quad \text{[RETURN]}$$

Workflow of SparseDroid



- Replace IFDS solver with Sparse IFDS solver
- On-demand SCFG Construction

Evaluation

- Theoretically same precision as FlowDroid
  - validated by DroidBench

Evaluation

- Theoretically same precision as FlowDroid
  - validated by DroidBench
- Research questions
  - **RQ1.** Is SparseDroid faster?
  - **RQ2.** Is SparseDroid more memory-efficient?
  - **RQ3.** Is the sparse IFDS algorithm effective?
  - **RQ4.** Is the on-demand SCFG construction effective?

Experimental Setup

- Benchmarks:
  - 34 apps from Fossdroid and 6 apps from Google Play

Experimental Setup

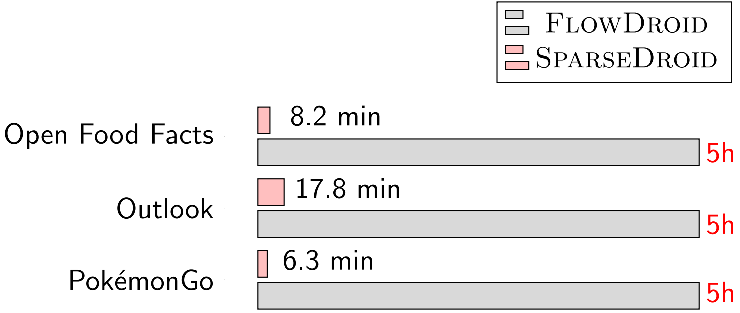
- Benchmarks:
  - 34 apps from Fossdroid and 6 apps from Google Play
- Platform:
  - Intel Xeon E5-1660 v4 CPUs (3.20GHz) server, 256GB RAM
  - Ubuntu 16.04.4 LTS (Xenial Xerus)
  - JVM: -Xmx220GB
  - IFDS solver time budget: 5 hours
  - 8 threads propagate facts

RQ1. Is SparseDroid faster?

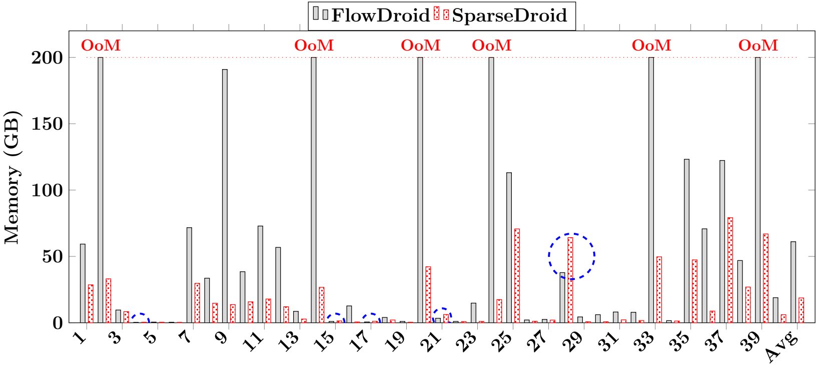
- 1.1x - 357.3x, 22.0x average speedups

RQ1. Is SparseDroid faster?

- 1.1x - 357.3x, 22.0x average speedups
- SparseDroid finishes analyzing within 18 min

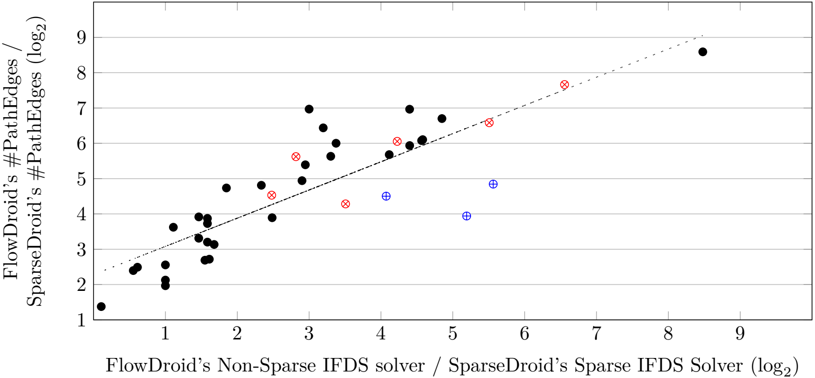


RQ2. Is SparseDroid more memory-efficient?



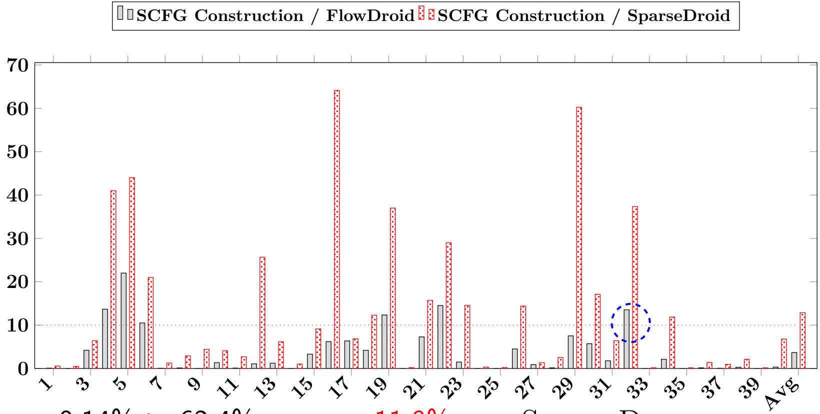
- FlowDroid runs out of memory for 6 apps
- Significantly reduce memory requirements
  - except the 5 apps marked in blue circles

RQ3. Is the sparse IFDS algorithm effective?



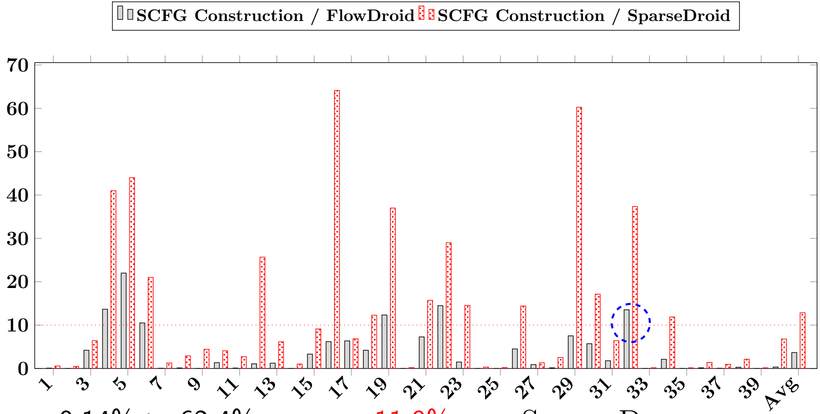
- Positive correlation between PathEdge reduction rate and speedups
- No correlation between app size and speedups

RQ4. Is the on-demand SCFG construction effective?



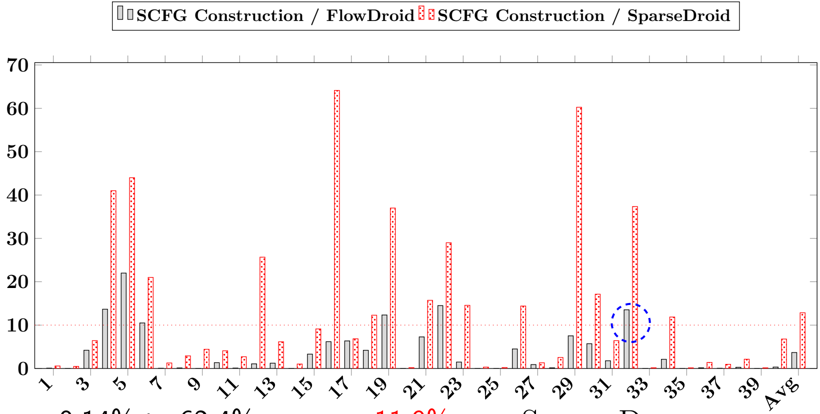
- 0.14% to 62.4%, average 11.9% over SparseDroid

RQ4. Is the on-demand SCFG construction effective?



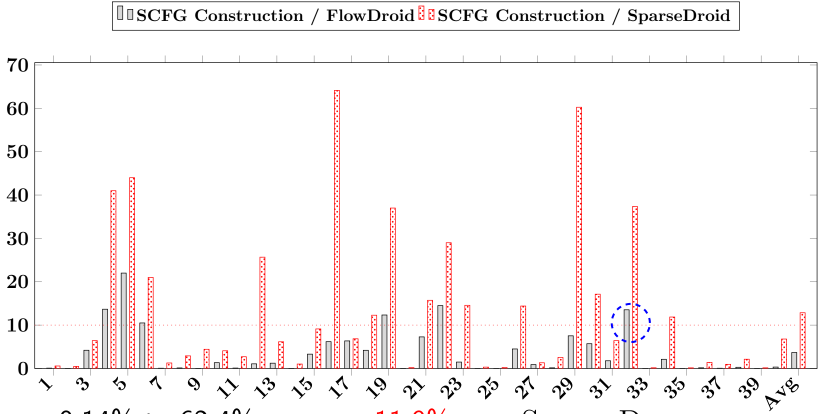
- 0.14% to 62.4%, average 11.9% over SparseDroid
- 0.0% to 29.0%, average 3.3% over FlowDroid

RQ4. Is the on-demand SCFG construction effective?



- 0.14% to 62.4%, average 11.9% over SparseDroid
- 0.0% to 29.0%, average 3.3% over FlowDroid
- the blue circle marked app uses 1.08s for SCFG but reduces analysis time from 8s to 2.9s (2.7x)

RQ4. Is the on-demand SCFG construction effective?



- 0.14% to 62.4%, average 11.9% over SparseDroid
- 0.0% to 29.0%, average 3.3% over FlowDroid
- the blue circle marked app uses 1.08s for SCFG but reduces analysis time from 8s to 2.9s (2.7x)
- Incurs overheads but brings more performance benefits

Summary

Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis

- Present a sparse IFDS algorithm.
  - send facts directly to their next use points according to SCFG.
- Present SPARSEDROID
  - a new taint analysis tool using sparse IFDS solver.
- Achieve a better performance
  - save time and memory

Q & A

Thanks!

backup slides

Exist app that needs more analysis time?

- Yes!
  - Very few applications take longer than FlowDroid.
  - Both FlowDroid and SparseDroid use less than 200 ms for these apps.
  - Analysis time is very close to each other.
  - We randomly select the benchmark and they are unluckily missed.

Why the 5 apps use more memory?

Category	App	FlowDroid (GB)	SparseDroid (GB)
Development	de.k3b.android.contentproviderhelper	0.3	**0.4**
Navigation	com.ilm.sandwich	1.0	**1.4**
Phone&SMS	opencontacts.open.com.opencontacts	0.6	**1.1**
Sci&Edu	com.ichi2.anki	3.4	**6.2**
System	com.github.axet.callrecorder	37.8	**64.2**

- SCFG cache does take some memory space but within the acceptable range
- Memory usage fluctuation exists for different run.

Why most apps in your benchmark are open-source ?

- 34 apps from FossDroid and 6 apps from Google Play.
- Xeon E5-1660 v4 CPUs (3.20GHz) server with 256GB RAM; Time Budget: 5 hours
- FlowDroid often terminate early or run timeout on real-world apps within our setting.
- Apps from FossDroid are generally smaller comparing with that from Google Play.

What is the difference of this work with other Sparse work?

- Different Analysis Clients
  - POPL'09 and CGO'11 for Pointer Analysis.
  - ISSTA'12 for memory leak dectection client.
  - this paper is for a kind of Data Flow Analysis called IFDS.
- Sparsification happens in different phase.
  - POPL'09, CGO'11 and ISSTA'12 do the sparsification in their pre-analysis.
  - in our work, sparsification and analysis happens simultaneously.
- Different Challenges
  - CGO'11 and ISSTA'12 : captures def-use chains for address-taken pointers.
  - this work skips edges of fact-specific identity flow function without breaking summary process of the original IFDS Algorithm and without losing the performance benefits reaped from its subsequent sparse propagation.

Is the Sparse technique for IFDS general?

- Fairly general
  - Pointer analysis (ECOOP'2016)
  - Typestate analysis (OOPSLA'2017)
  - Constant propagation (TCS'1996)
  - Uninitialized variables (POPL'1995)
  - Android compatibility detection (ASE'2018)
  - Taint analysis (PLDI'2014, ICSE'2015)
- We present SPARSEDROID built on top of FLOWDROID just for evaluation.

Interprocedural, Finite, Distributive, Subset (IFDS)

- IFDS: $IP = (G^*, D, F, M, \sqcap)$
- $G^* = (N^*, E^*)$ is a supergraph
  - $\{G_1, \cdots, G_n\}$, G_{main} stands for the main function
  - unique start node s_p, exist node e_p for G_p
  - each call represented by a call-node c and a return-site node r
  - Normal Edges, Call Edges, Call-to-Return Edges, Return Edges
- D is a finite set
- $F \subseteq 2^D \to 2^D$ is a set of distributive functions
- $M : E^* \to F$ is a map from G^*'s edges to dataflow functions
- $\sqcap$ is either union or intersection

Optimization of IFDS

- Reps et al. (POPL'1995), $O(ED^3)$ time, $O(ED^2)$ space
  - dense propagation of facts (many dots)
  - time- and memory-intensive when E and D are large
- Memory-efficient implementation in WALA
  - encode D with bit-vector, save memory
- Multi-threaded implementation
  - Arzt et al. (PLDI'2014), Bodden et al. (SOAP'2012), and Naeem et al. (CC'2010)
  - utilize multiple processors, save time
- We use Sparsification