Practical DIFC Enforcement on Android

Adwait Nadkarni[1], Benjamin Andow[1], William Enck[1], Somesh Jha[2]

[1]North Carolina State University

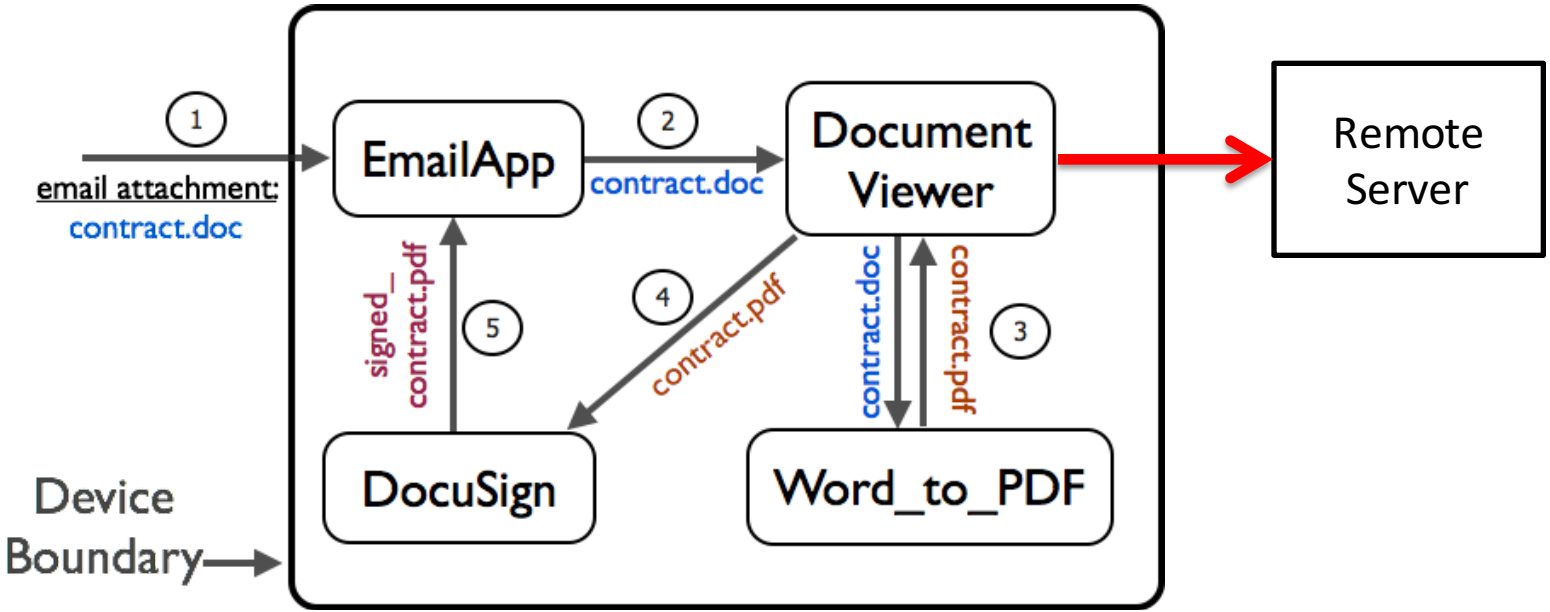
[2]University of Wisconsin-Madison

The “new” Modern Operating Systems



1. Applications are *security principals.*
2. Applications *share* data.

Example use case – Email



How do we enable data sharing among apps, and also prevent ***unauthorized disclosure****?*

The Problem

➢ The problem stems from the *loss of control* over the flow of shared data.

- So the solution is … *Information Flow Control (IFC)*
- For data secrecy, the *Bell-LaPadula* model.
- IFC models have only seen limited use (e.g., military).
- *Centrally administered policy* – What about application-specific user data (e.g., email attachment).

Decentralized Information Flow Control (DIFC)

- Secrecy for *application-specific* user data.
- Applications are "Data Owners" that define the secrecy policy for their own data.
- Data Owners can
  - *create* new *labels* (i.e., security classes)
  - *define* the policy for their labels
  - *declassify* their labels
- E.g., the Email app creates label *{email}*.

DIFC for Android

- DLM-like DIFC policy (*Aquifer*) [Nadkarni & Enck, CCS 2013]
- Noninterference proof [Jia et al., ESORICS 2013]
- Storage-level enforcement primitives (*Maxoid)* [Xu & Witchel, EuroSys 2015].

Takeaway – Hard to make DIFC enforcement both *secure* and *backwards compatible* with *unmodified* legacy applications on Android.

DIFC Enforcement Challenges on Android

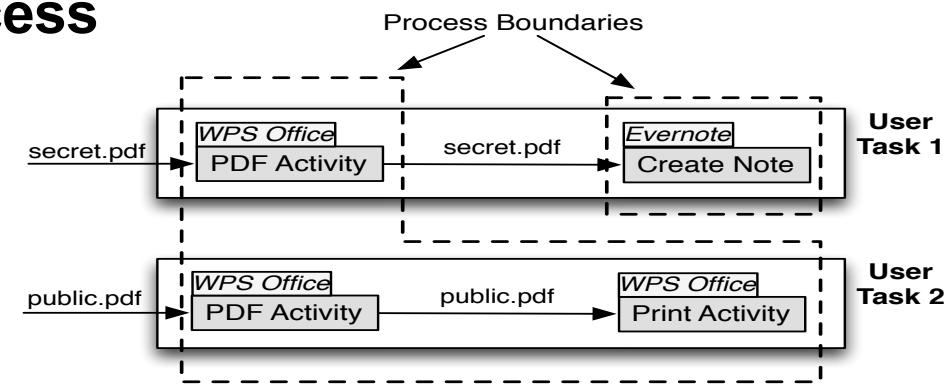
1. Subject granularity / labeling precision.
2. Label change and propagation.
3. Network Declassification.

C1. Subject Granularity

1. **Fine-grained PL variable** (e.g., TaintDroid [Enck et al.], Laminar [Roy et al.])
   - False Negatives (Implicit flows)

```
//'a' contains a secret
b = false;
if (a > 10) {
    b = true;
}
```

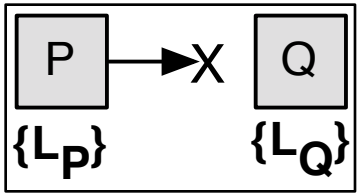
2. **Coarse-grained OS Process** (e.g., Aquifer, Jia et al.'13)
   - False positives: (Multiple tasks in one process)



Is there a balance?

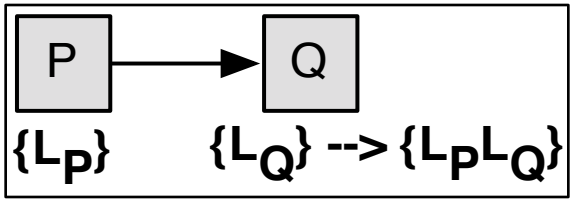
C2. Label Propagation

A. ***Explicit flows*** (e.g., HiStar [Zeldovich et al.'06], Flume [Krohn et al.'07])
   - *Limitation:* Unsuitable for ad-hoc communication (e.g., unpredictable user-directed sharing).



A. Explicit labels

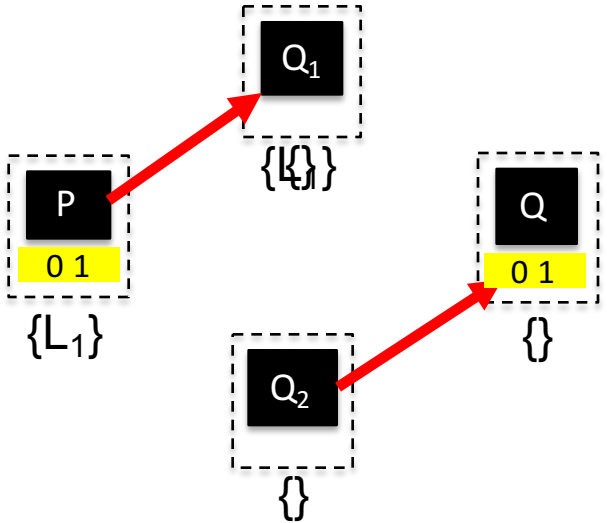
B. ***Floating labels*** (e.g., Asbestos [Efstathopoulos et al.'05]).
   - Communication is always enabled.
   - Are floating labels secure and practical?



B. Floating labels

C2. Label Propagation

- Limitations of floating labels:
  1. **Security:** Implicit data leaks. [Krohn and Tromer'09]



Prior Agreement:

1) Q_1 and Q_2 will call Q at a predefined time
   - ***Unless***, they get called by P.
2) P will call Q_i if i^{th} bit is 0.

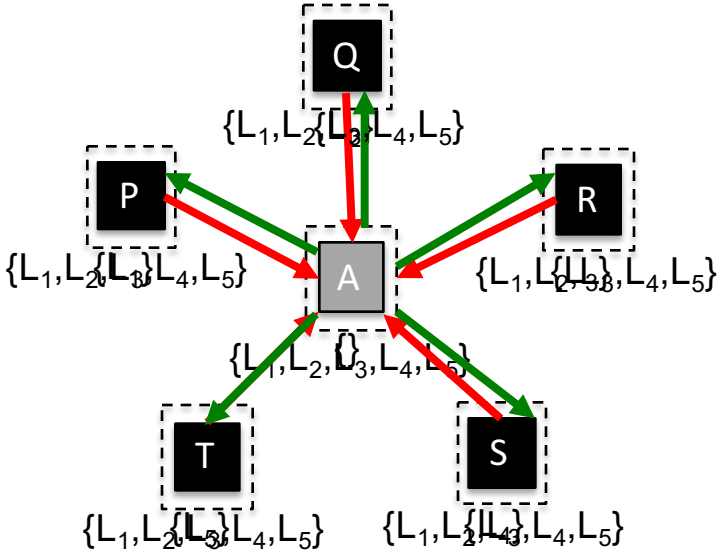
Therefore, the Q_i that calls Q back must indicate 1!

Step 1: P calls Q with data '0 1'

C2. Label Propagation

- Limitations of floating labels:

2. Practicality: Label Explosion.



Scenario: *Subject "A" reads/writes to many other subjects (with different labels)*

"A" could be:

1) an Android service component.
2) an Android content provider component.
3) a shared file (e.g., SharedPreferences).
4) A general-purpose application (e.g., PDF reader); due to *application multi-tasking.*

C3. Network Declassification

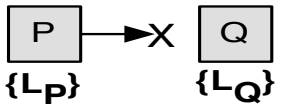
- The network is public.
- *Declassification* is necessary before network export.
- Existing DIFC policy allows
  1. the data owner to declassify data, or
  2. specify the security principal that can declassify data.
- This is not practical on Android, where
  1. The environment is *network driven*.
  2. The user may not ideally be limited to using a few applications for export.

Weir

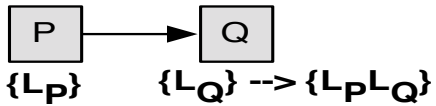
- **Secure** and **backwards compatible** DIFC Enforcement for Android.
- *Lazy* Polyinstantiation – Making floating labels context sensitive. (C1, C2)
- *Domain Declassification* – Providing an alternate network declassification primitive. (C3)

Lazy Polyinstantiation

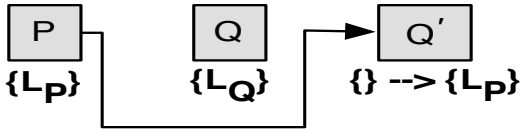
- **Polyintantiation** – Creating multiple (context-sensitive) instances.



Explicit labels



Floating labels

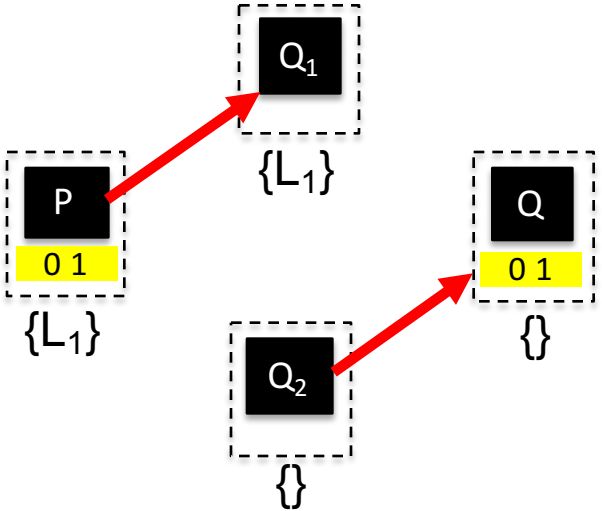


Floating labels w/ polyinstantiation

- **What makes it *Lazy* ?**
  1. Event-driven –
     - Caused by a call (i.e., Inter Component Communication).
  2. Keeps track of existing instances –
     - Creates new instance only if one in a matching secrecy context is unavailable.

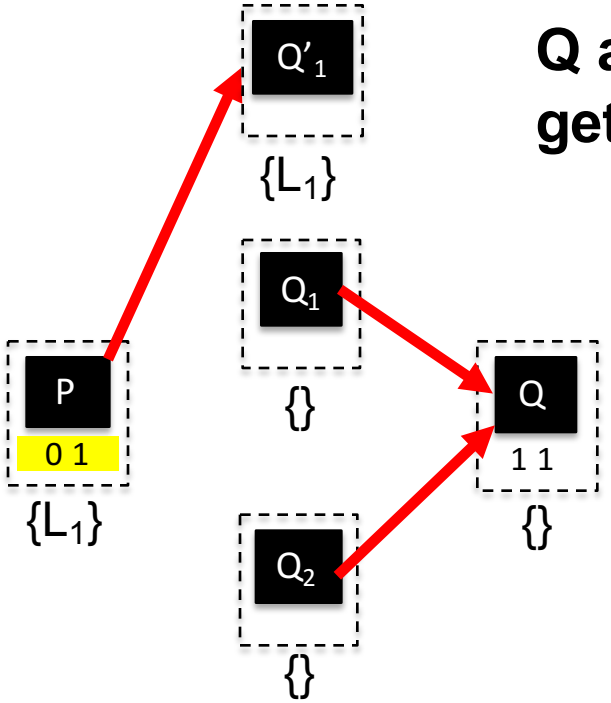
Polyinstantiation – Preventing implicit flows

- Recap: Floating labels, Q gets original data 01.



- With polyinstantiation:

Q always gets 11.

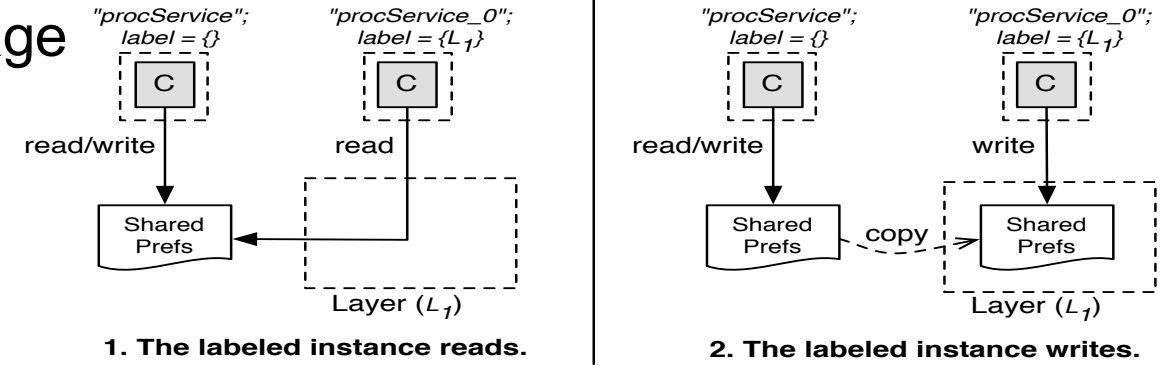


Practicality of Polyinstantiation

1. **No label explosion:**
   - In Weir, labels do not float in the traditional sense.
   - Context-specific instances of shared components prevent label explosion.
2. **Process-level labeling:**
   - Context-sensitivity eliminates false positives.

Design of Polyinstantiation on Android

- *Polyinstantiation in memory:* Processes and components
  1. Polyinstantiate if needed based on the caller's label.
  2. Maintain process-component mapping (backwards compatibility).
- *Polyinstantiation on storage:* Context-specific layers.
  - Transparent Storage access.
  - Copy-on-write



Why this works?

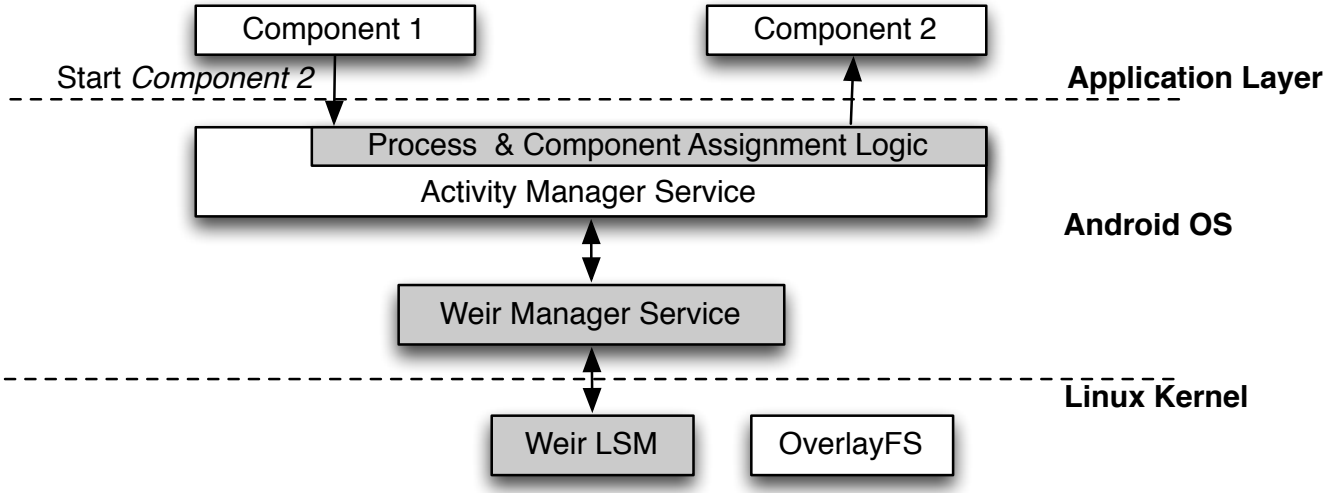
1. **Component Model** – *Activity* (UI), *Service* (Daemon), *Provider* (Data Interface/Daemon), *Receiver* (Event handler).
2. **Communication** –
   a. Indirect – *Activity Manager Service*, predefined API (e.g., startActivity, bindService).
   b. **Direct** – *Binder*; exchange Binder objects first (e.g., RPC to a service, first bound using bindService*).*

Domain Declassification

- **Intuition:** In a network driven environment, it may be more practical to reason about *where* the data is being delivered, rather than *who* is performing the export.
- Data Owner (i.e., security class creator) associates a set of *trusted network domains* with the security class.
  - Set $\mathbf{t^D}$ for a security class **t**.
- Enforcement
  - For export to a domain **d**, if $\mathbf{d} \in \mathbf{t^D}$, **t** can be implicitly declassified.

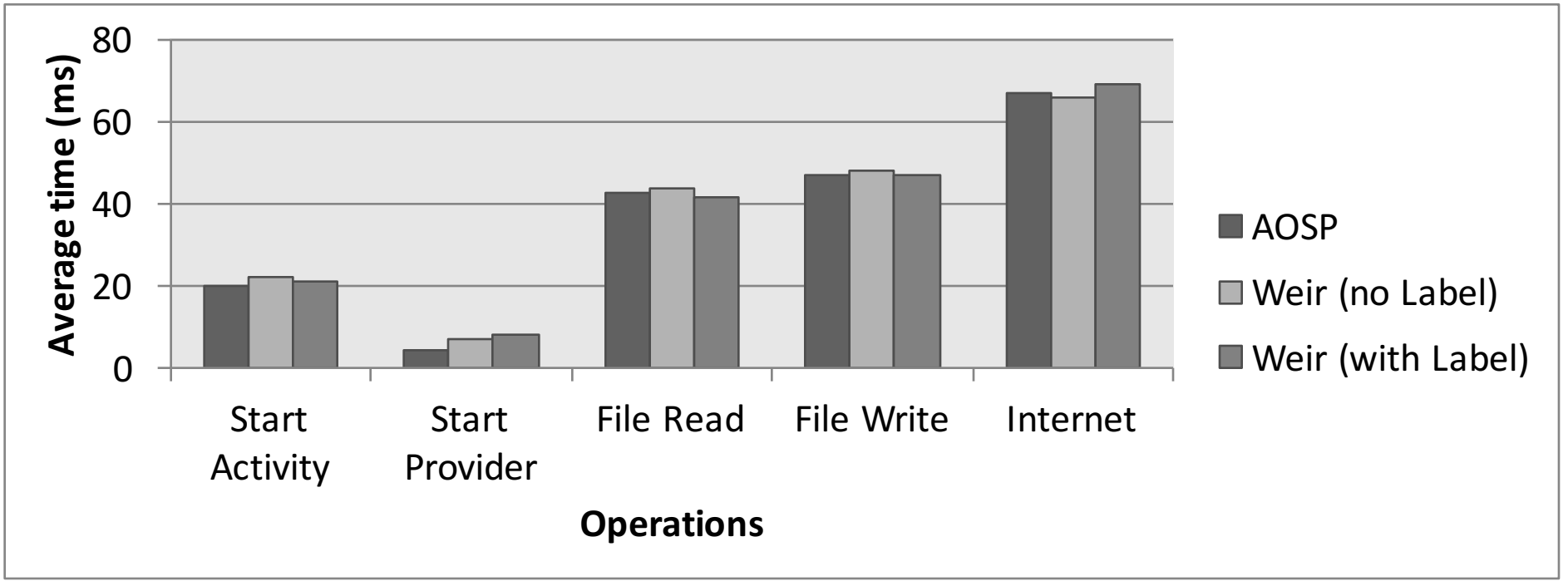
Implementation & Performance

- Implemented on Android v5.0.1, kernel v3.4
- Policy Model – Flume [Krohn et al.]
- Source code available at http://wspr.csc.ncsu.edu/weir/



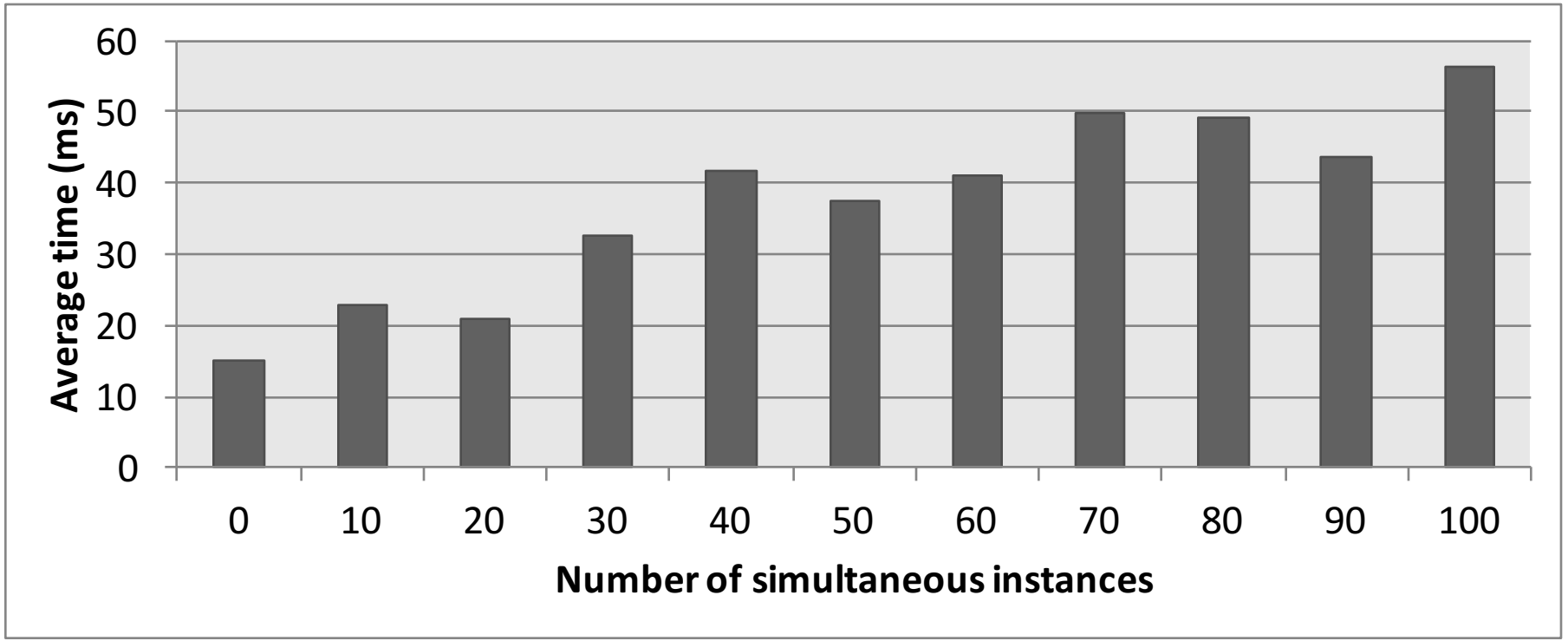
Evaluation: Microbenchmarks

- Micro-benchmarks for common operations:



Evaluation: Scalability

- Start 0-100 instances of an already started component:



Case Study – *K9 Email*

- **Objective:** Separation of the personal and work email.
- **Applications:**
  - *BCloud* app, allows the user to sync her work cloud data (e.g., contacts, documents) to the device.

1. Creating BCloud's tag

```
//Creating a tag 't'
domains = {"www.bcloud.com",
"smtp.bcloud.com",...};
createTag("t", domains);
```

2. Using BCloud's tag

```
// Add ''t'' to the intent's label.
 intent.addToLabel(''t'');
// Add data to the intent
startActivity(intent); //Call
```

  - Unmodified *K-9 Email* app, used for both the personal (*smtp.gmail.com*) and work (*smtp.bcloud.com*) email.

Case Study – *K9 Email*

- **Steps –**
  1. Share BCloud data with K9 Email (*BCloud* applies tag '*t*').
  2. Add contacts, attach a document
  3. Switch to home screen.
  4. Share unlabeled data with K9 Email, and repeat step 2 (now in the personal context).
  5. Send emails from both contexts.

Case Study – *K9 Email*

- **Observations**
  - **General:** Instances of *K9 Email* existed simultaneously in both contexts.
  - In enterprise context ‘t’:
    - **Files and Contacts:** Data from personal context + the enterprise context ‘t’ is visible.
    - **Network Export:** K-9 Email could only sync with the enterprise account’s servers.

Caveats

- **Centralized perspective:**
  - Labeled + unlabeled files can only be seen in the specific labeled context (e.g., *t*).
  - Files with different non-default labels can only be seen via a trusted OS application that can accumulate all labels.
- **Updates to default layer:**
  - Updates to contacts in the personal context are not visible in other contexts, once copied.

Closing Remarks

- *Weir* makes floating label enforcement possible on Android via lazy polyinstantiation.
- Some open challenges remain:
  - Centralized perspective over labeled data.
  - Propagation of changes made to the default layer.
  - Instance explosion.
  - Defining trusted domains.

Thank you!

Adwait Nadkarni

anadkarni@ncsu.edu

http://wspr.csc.ncsu.edu/weir/

Looking for a position starting Fall 2017.