A Characterization of State Spill in Modern OSes

Kevin Boos Emilio Del Vecchio Lin Zhong

ECE Department, Rice University

EuroSys 2017

Linux kernel map

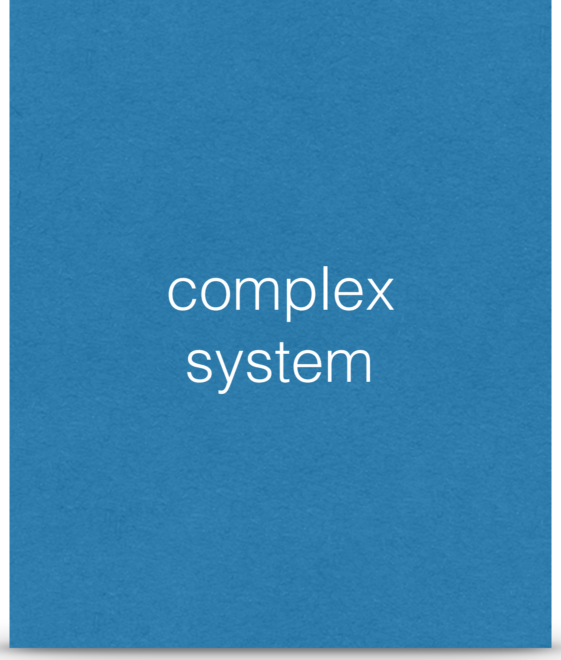
functionalities | human interface | system | processing | memory | storage | networking

layers: user space interfaces | virtual | bridges | logical | device control | hardware interfaces | electronics

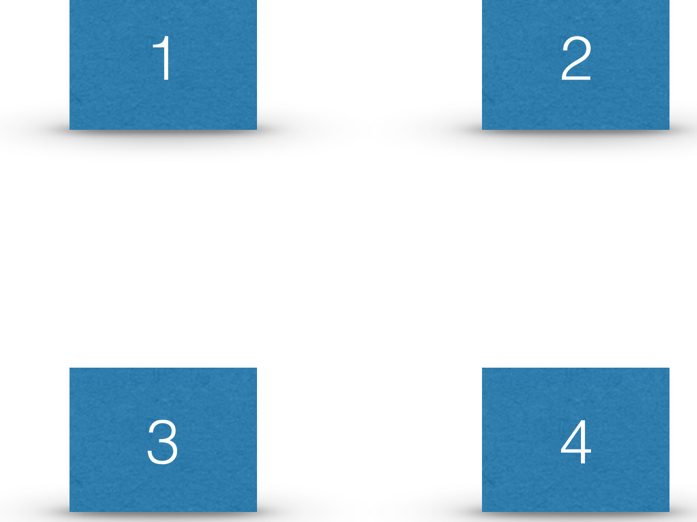
How do we deal with complexity?

© 2007, 2010 Constantine Shulyupin www.MakeLinux.net/kernel_map

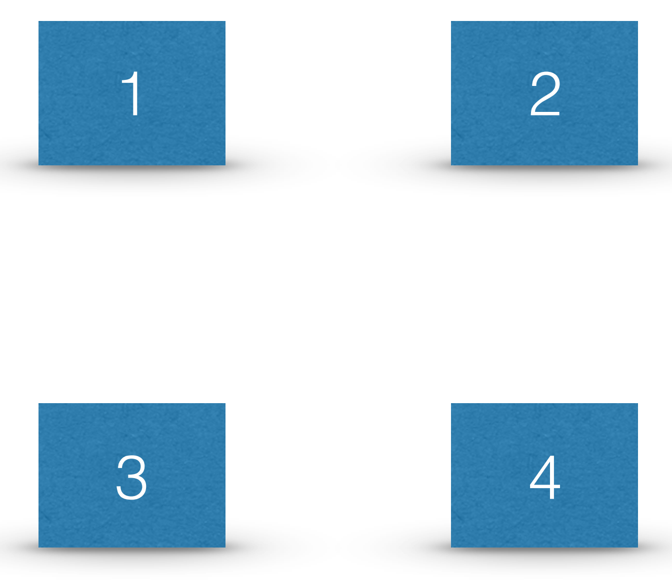
Modularization



Modularization

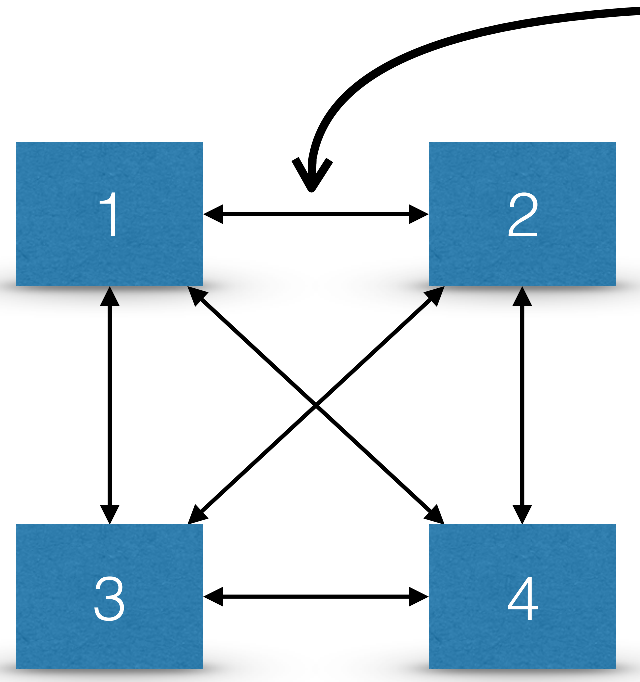


Reducing complexity *should* make things easier…



- Process migration
- Fault isolation & fault tolerance
- Live update, hot-swapping, software virtualization
- Maintainability
- Security and more

Modularization is not enough!



interactions have complex effects!

Effects of interactions:

- Propagation of data and control
- Changes to the state of each entity

state spill

State spill in a nutshell

a new term to describe the phenomenon when:

A software entity's state undergoes **lasting change** as a result of an interaction with another entity.

Outline of contributions

1. Define and identify state spill as a root cause of challenging problems in computing
2. Classify state spill examples collected from real OSes
3. Automate state spill detection with StateSpy
4. Results from Android system services

Definition of State Spill

State spill definition by example

Source (application)

```
public void main() {
  int id = ⬟;
  byte cfg = ▲;
  fn cb = handleCb; ★

  service.addCallback(
    id, cfg, cb);

  log("added cb!");
}


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

Destination (system service)

```
public class SystemService {
  static int sCount;
  byte mConfig;
  List<Callback> mCallbacks;
  int unrelated;

  public void addCallback(
        int id, byte cfg,
        Callback cb) {
  ⬟ int b = id;
    Log.print("id=" + b);
  ▲ mConfig = cfg;
  ★ mCallbacks.add(cb);
  ◆ sCount++;
  }
}
```

temporary

Before

(empty)

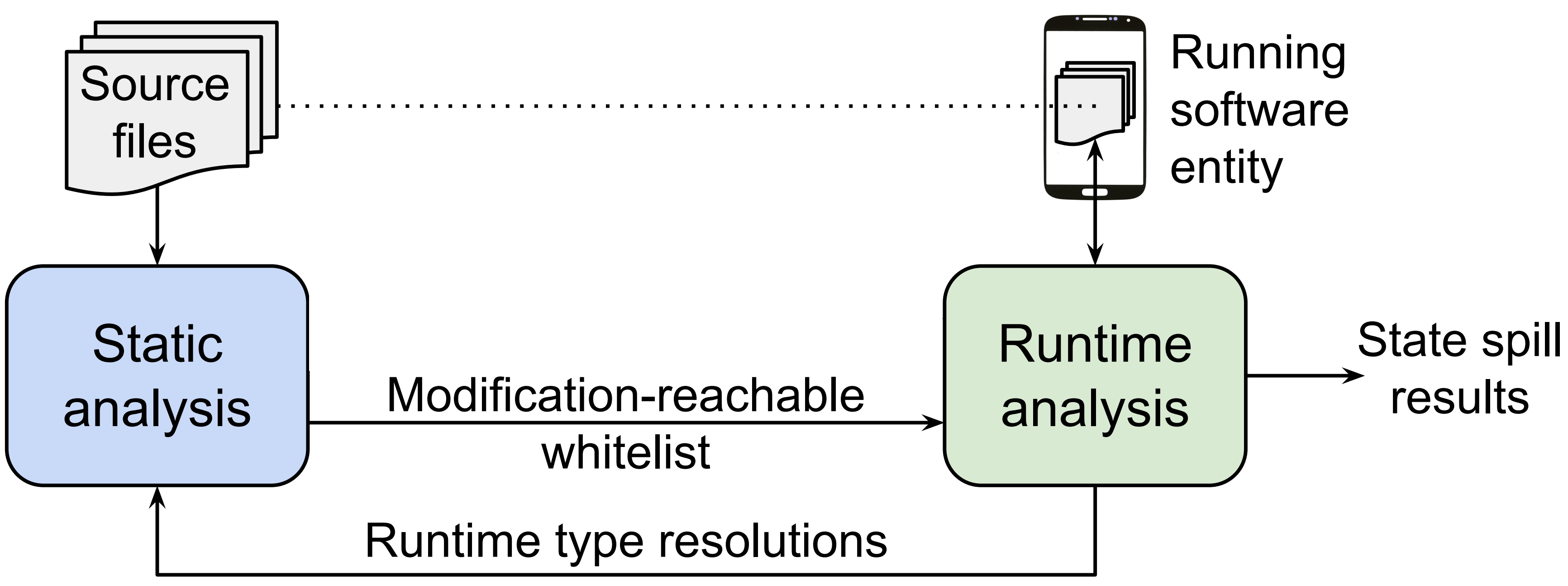
During

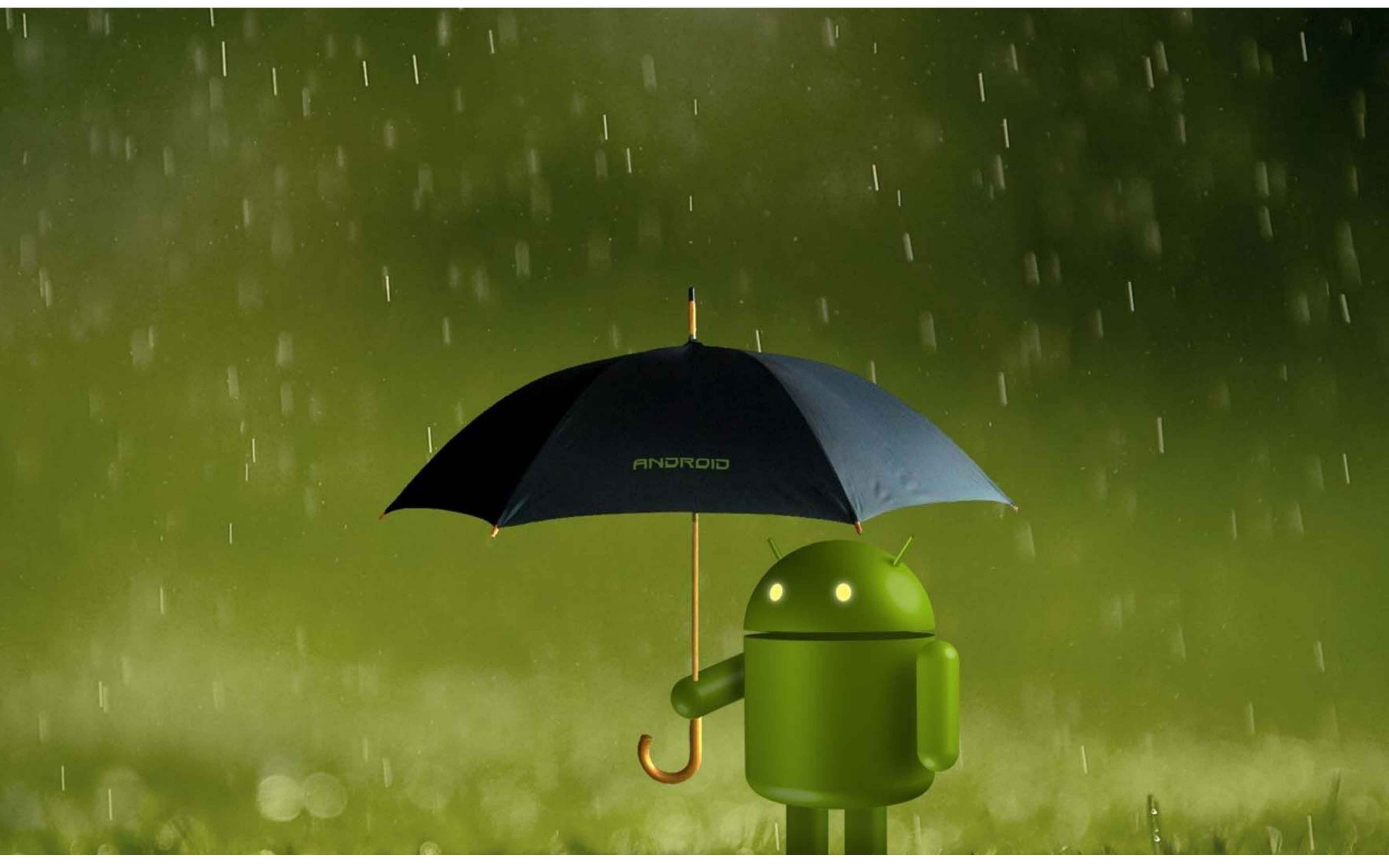
After

StateSpy: Automated State Spill Detection

STATESPY: runtime + static analysis

- Goal: help developers understand how state spill occurs in their entities





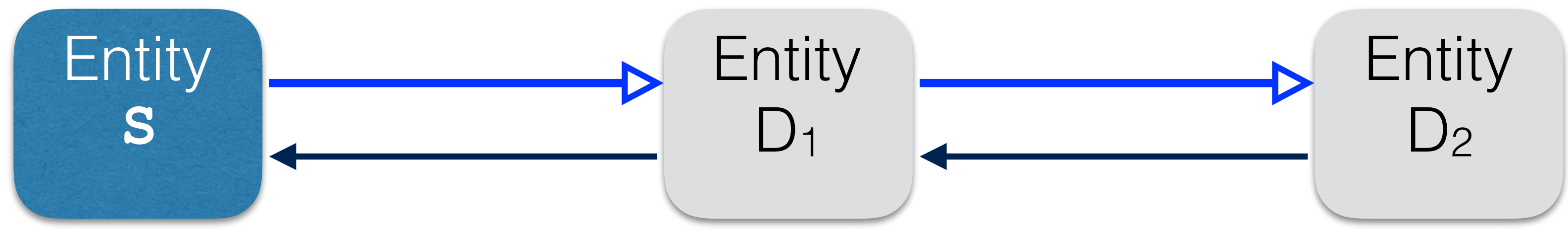
State Spill in Android Services

Evaluating Android system services

- StateSpy monitors service stub boundary (`onTransact`)
- `monkey` induces real apps to invoke various transactions

Found state spill in 94% of service stubs analyzed.

Secondary state spill

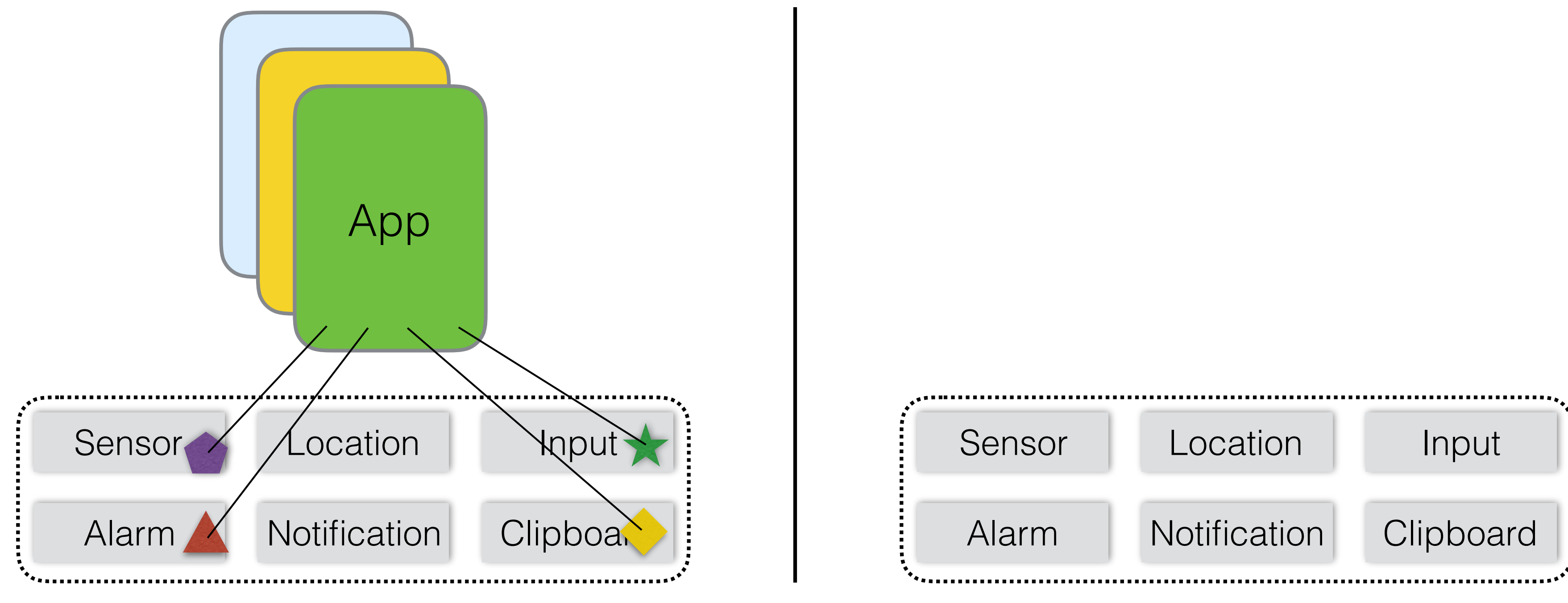


User Applications

UiModeManagerService
VibratorService
KeyguardService
AlarmManagerService
UsbService
AudioService
InputManagerService
ActivityManagerService
HdmiControlService
NotificationManagerService
DisplayManagerService
PowerManagerService
UserManagerService
StatusBarManagerService
WindowManagerService
SensorService
PackageManagerService

Hinders fault tolerance, hot-swapping, maintainability

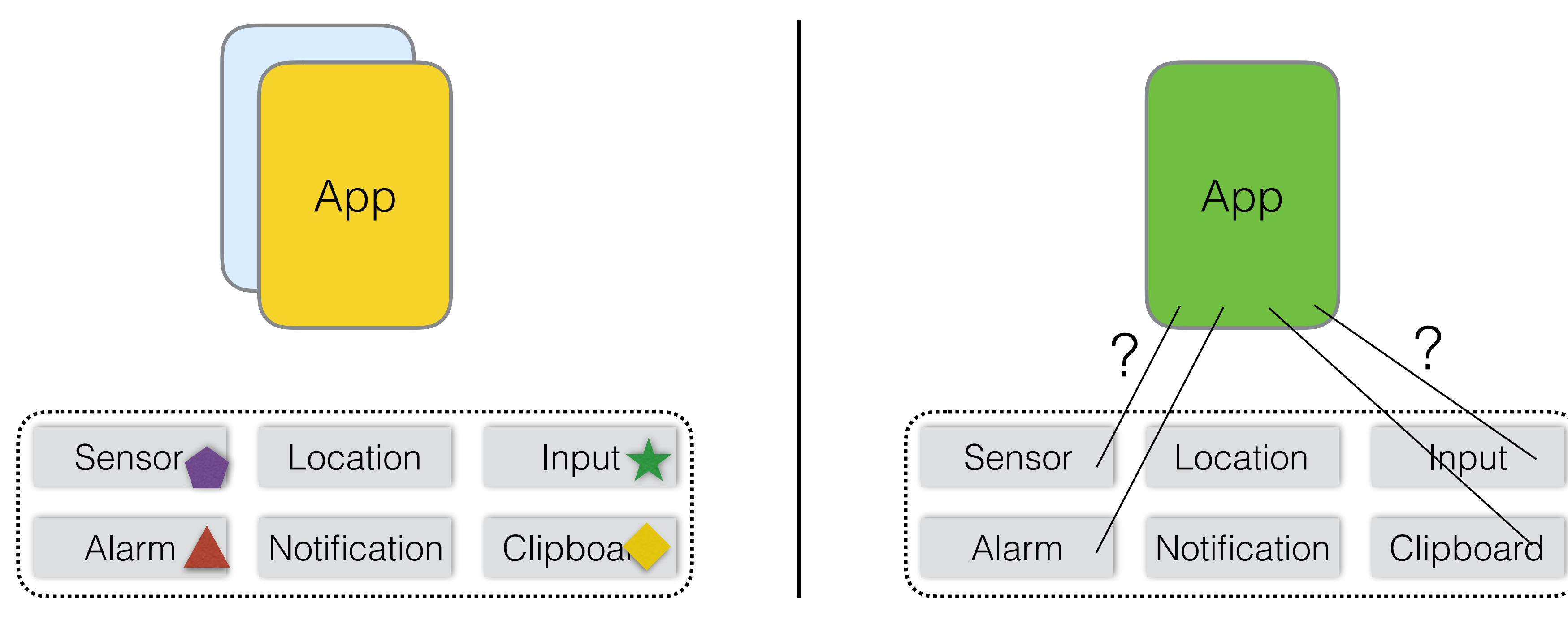
Case study: Flux [EuroSys’15]



- Android app migration

Alex Van’t Hof, et al., “Flux: multi-surface computing in Android”, EuroSys 2015.

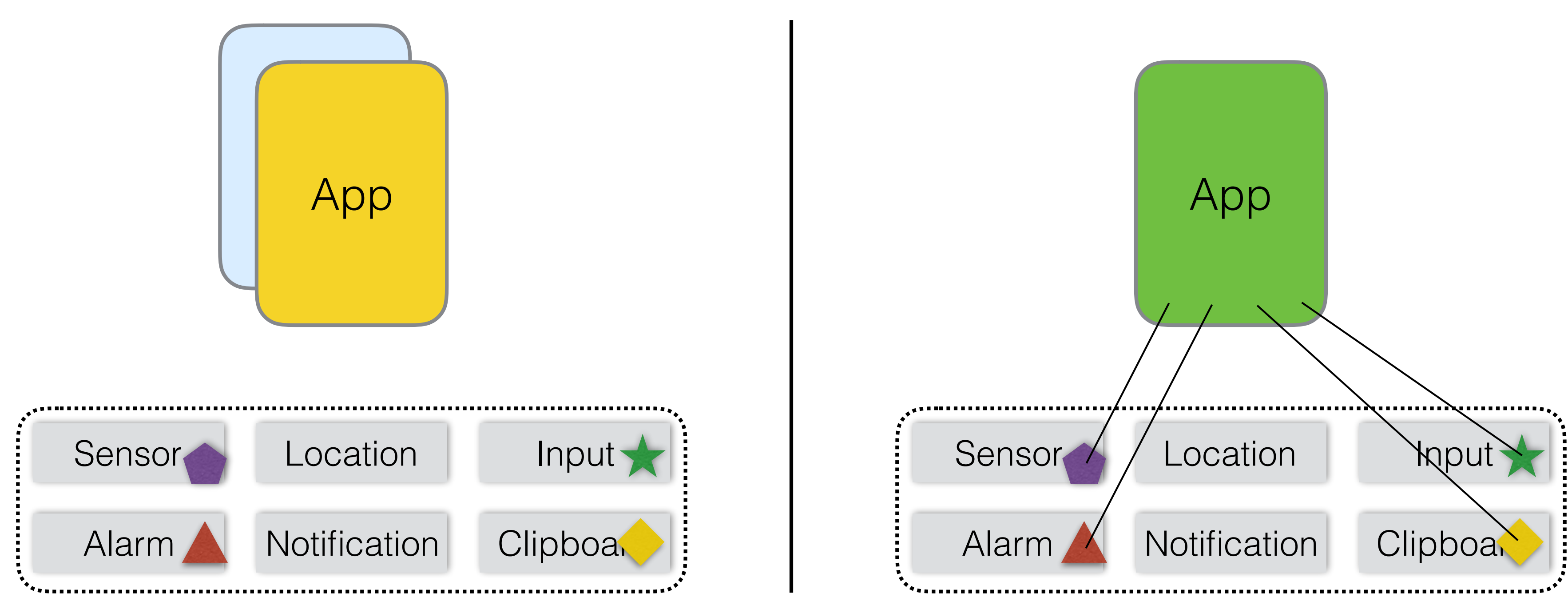
Case study: Flux [EuroSys'15]



- Android app migration

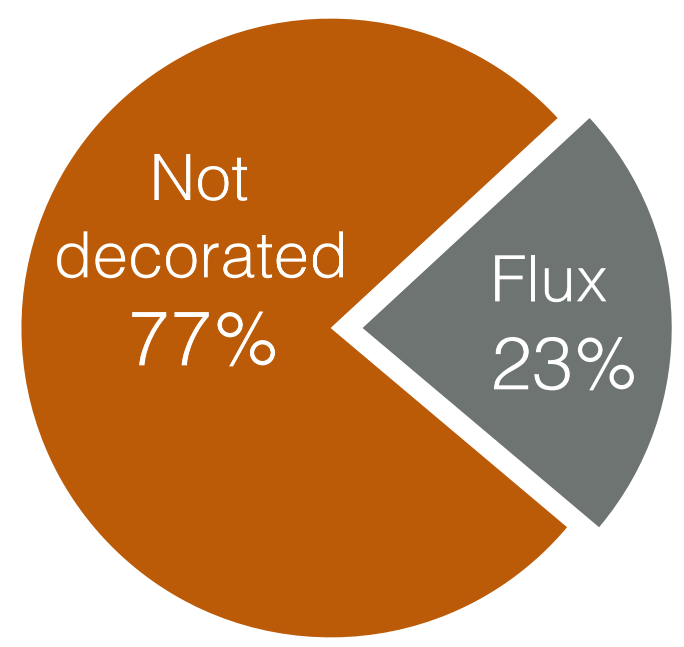
Alex Van't Hof, et al., "Flux: multi-surface computing in Android", EuroSys 2015.

Case study: Flux [EuroSys'15]



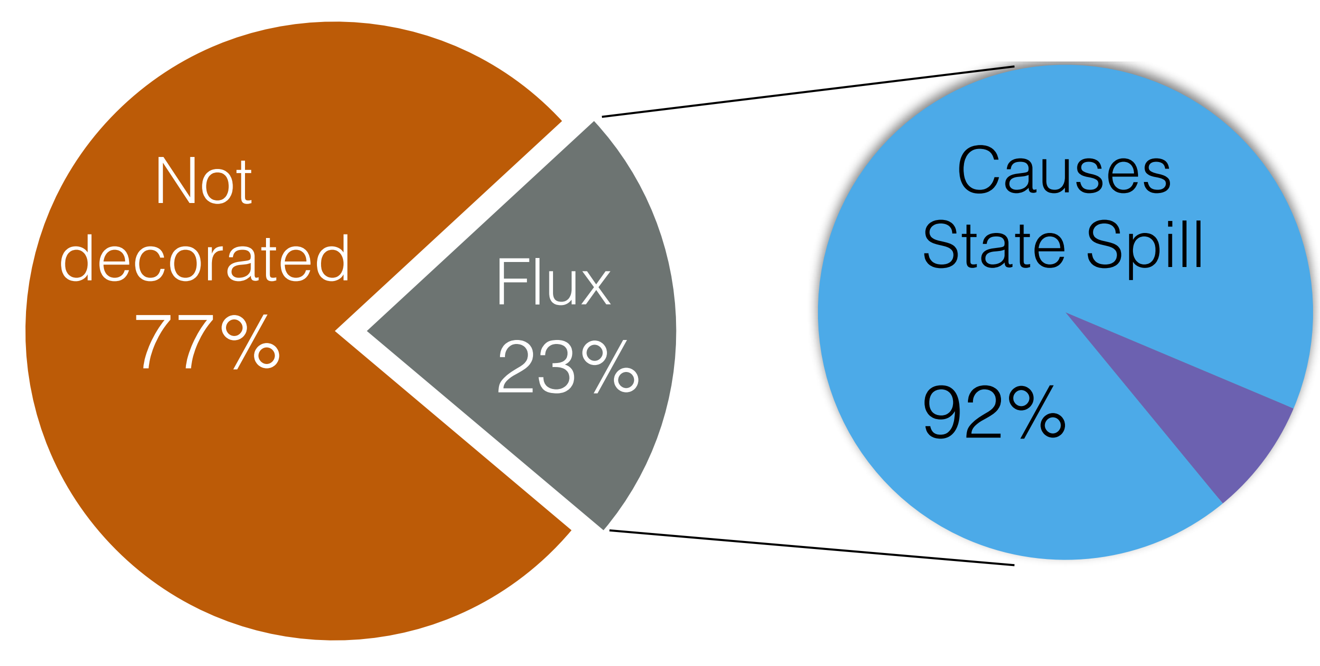
- Android app migration via record & replay
- Manually handles residual dependencies with *decorator methods* for each service transaction
  - Significant effort to overcome state spill

Comparison with Flux



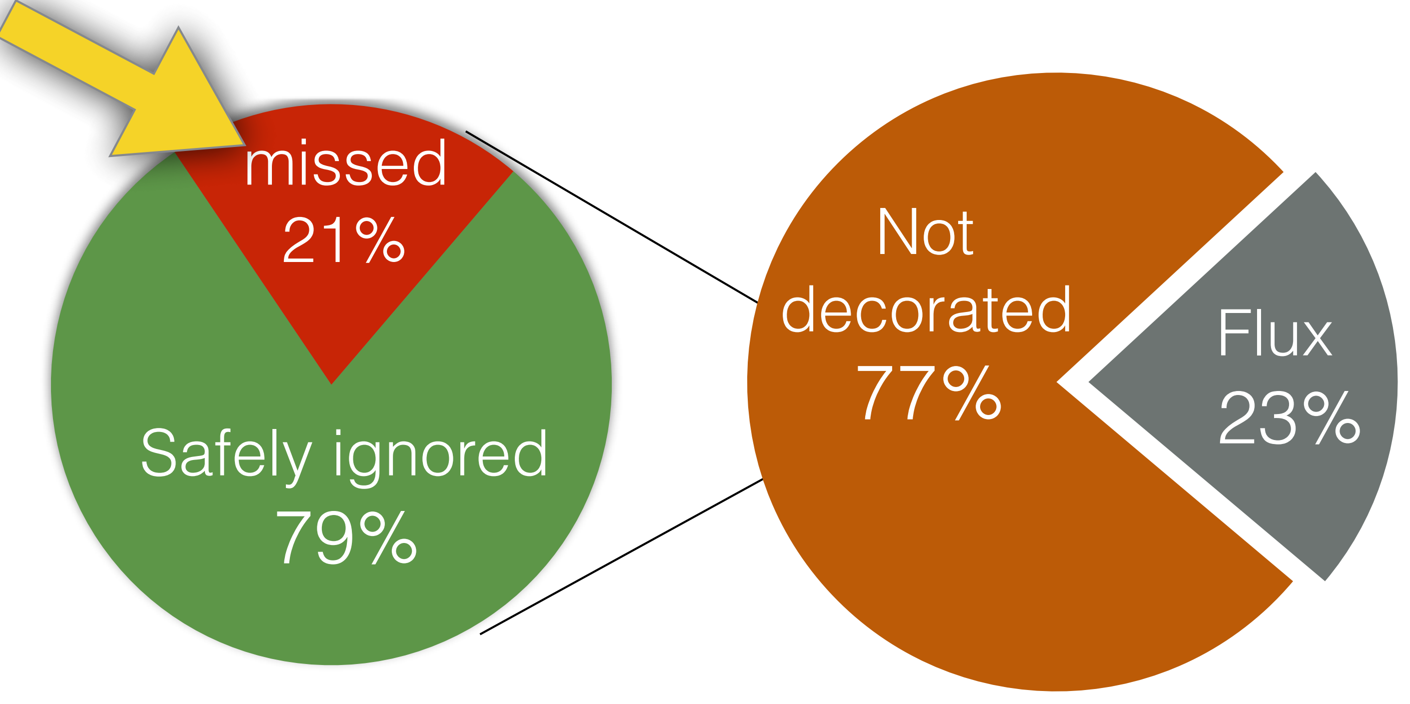
- Using Flux apps, we reproduced 113 unique transactions for analysis with STATESPY

High correlation with state spill



- State spill identifies problematic service transactions
  - and which states need special handling

STATESPY catches what's missing



- Found state spill in 18 (21%) undecorated methods, each is potentially dangerous
- Easy detection demonstrates STATESPY's utility

Parting Thoughts

Designs to avoid state spill

- Client-provided resources
- Stateless communication RESTful principle
- Separation of multiplexing from indirection
- Hardening of entity state
- Modularity without interdependence

Related work

- Coupling[1]/modularity[2] as a *necessary* condition
- Info-flow analysis[3,4]
- Designs that partially reduce state spill
    - Compartmentalizing important states
        - Barrelfish/DC[5], Microreboot[6], CuriOS[7]
- RESTful architectures (web)[11,12]

Conclusion

- State spill is an underlying problem that hinders many computing goals
- Prevalent and deeply ingrained in many OSes
- Reducing state spill will lead to better designs
  - More so than minimizing coupling, etc.
- Next steps: redesign OS to minimize state spill

STATESPY & more: http://download.recg.org

References

(1) J. Offutt, et al., “A software metric system for module coupling.” *Journal of Systems and Software*, 1993.

(2) B. Ford, et al., “The Flux OSKit: A substrate for kernel and language research.” *SOSP*, 1997.

(3) S. Arzt, et al., “FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps.” *PLDI*, 2014.

(4) W. Enck, et al., “TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones.” *OSDI*, 2010.

(5) G. Zellweger, et al., “Decoupling cores, kernels, and operating systems.” *OSDI*, 2014.

(6) G. Candea, et al., “Microreboot - a technique for cheap recovery.” *OSDI*, 2004.

(7) F. David, et al., “CuriOS: Improving reliability through operating system structure.” *OSDI*, 2008.

(8) D. Engler, et al., “Exokernel: An operating system architecture for application-level resource management.” *SOSP*, 1995.

(9) D. Porter, et al., “Rethinking the library os from the top down.” *ASPLOS*, 2011.

(10) A. Madhavapeddy, et al., “Unikernels: Library operating systems for the cloud.” *ASPLOS*, 2013.

(11) C. Pautasso and E. Wilde. “Why is the web loosely coupled?: a multi-faceted metric for service design.” *WWW*, 2009.

(12) C. Pautasso, et al., “RESTful web services vs. ’big’ web services: making the right architectural decision.” *WWW*, 2008.