A Characterization of State Spill in Modern Operating Systems

Kevin Boos, Emilio Del Vecchio, and Lin Zhong
Rice University
{kevinaboos, edd5, lzhong}@rice.edu

Over the past few decades, operating systems research has gone to great lengths to achieve a spectrum of advanced computing goals: process migration, fault isolation and tolerance, live update, hot-swapping, virtualization, security, general maintainability, and more. Better *modularization*, in which related functionality is grouped into bounded entities, is often touted as the most apt solution for realizing such goals, and it seems promising from an initial glance. However, we argue that modularization alone is not enough, and that the *effects of interactions between modules* have a more pronounced impact on these goals, as shown below.

Many efforts towards these goals have been ad-hoc and platform-specific due to the complex nature of how software states change and propagate throughout the system, showing that modularity is necessary but insufficient. For example, several process migration works have cited "residual dependencies" that remain on the source system as an obstacle to post-migration correctness on the target system. Fault isolation and fault tolerance literature has long realized the undesirable effects of fate sharing among software modules as well as the difficulty of restoring a process or whole machine statefully after a failure, due to the sprawl of states spread among different system components. Live update and hot-swapping research has long sought to overcome the state maintenance issues and inter-module dependencies that plague their implementations when transitioning from an old to a new version, forcing developers to write complex state transfer functions. Likewise, state management poses a problem to the virtualization of arbitrary software components, significantly complicating the multiplexing and isolation logic of the underlying virtualization layer (e.g., kernel or runtime).

State Spill is the Underlying Problem

Our primary contribution is to identify the problem of *state spill* as a root cause of these problems and show that it is the underlying reason why many computing goals are so difficult to realize, even in the face of proper modularization. State spill is the act of one software entity's state undergoing lasting changes as a result of its interaction with another entity. For example, an application that interacts with a system service and causes it to store application-relevant states in its memory, then state spill has occurred from application to service.

The second contribution of this work is a manual analysis of state spill across a broad variety of real-world systems and their detrimental impact on the aforementioned computing goals. We find that state spill is often a byproduct of applying the *locality principle* to OS design, in that it often stems from design choices that favor programming convenience or performance over adherence to strict architectural or modularity principles. Based on these case studies, we establish a classification of state spill according to common entity design patterns: indirection layers, multiplexers, and communication facilitators.

Our third contribution is the StateSpy tool that automates the detection of state spill in real systems, and an in-depth, tool-guided analysis of state spill in Android's system service entities. StateSpy employs both runtime and static analysis techniques that automate the capture, inspection, and differencing of a software entity's state, with current support for Java environments. Our key insight for runtime analysis is to leverage existing *debugging frameworks* to non-intrusively capture and compare entity states, a technique that forgoes environment-specific features and generalizes to any execution environment. The static analysis component of StateSpy provides relevancy filters to the runtime component as part of a cooperative feedback loop that iteratively improves the output of each component. Guided by StateSpy, our experimental analysis of Android system services finds that harmful state spill is both prevalent and deep: nearly all Android system services have state spill, and it often occurs in a chain across multiple services.

The effects of state spill can be mitigated by rethinking existing software designs patterns and communication schemes. Modularization is not enough: reducing the impact of state spill is key to simplifying software components and making them amenable to migration, live update, fault recovery, virtualization, and other computing goals. We have investigated how state spill mitigation techniques like state hardening can reduce fate sharing in Android system services in order to make them fault tolerant and statefully recoverable.

In summary, we identify and formally define the problem of state spill in modern OSes, classify its various incarnations, and provide a tool that assists in the discovery of complex state spill incarnations. The nature of this work is not to offer a complete picture of state spill in every OS, but rather to highlight the existence of state spill as a harmful phenomenon and emphasize its negative impact on the realization of many modern computing goals.

Characterizing State Spill in Modern Operating Systems

Kevin Boos, Emilio Del Vecchio, and Lin Zhong

Advanced OS goals are challenging

Goal in OS literature	Impediments to that goal
Process migration	Residual dependencies on original system
Fault isolation/tolerance, software virtualization	Sprawl of states introduces fate sharing, complicates isolation & multiplexing logic
Live update and hot-swapping	Cannot modify individual entity in isolation; state transfer functions are non-trivial
Maintainability	Coupling remains despite modularization
Security	Loss of control over propagated data

State spill is the underlying cause

State spill is the act of a software entity's state **undergoing a lasting change** as a result of handling a transaction from another entity.

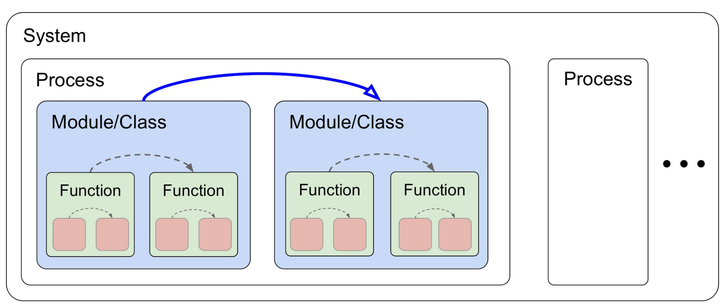
```
public class SystemService {
  static int sCount;
  byte mConfig;
  List<Callback> mCallbacks;
  int unrelated;

  public void addCallback(int id,
      byte cf, Callback cb) {
    int b = id;
    Log.print("id=" + b);
    this.mConfig = cf;
    this.mCallbacks.add(cb);
    sCount++;
  }
}
```

This method is a transaction handler invoked by application processes.

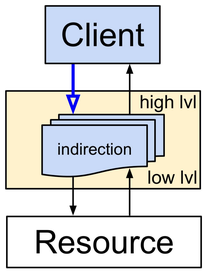
Entity granularity dictates state spill

State spill is relative to the chosen entity granularity. Low-level entity interactions (shaded) are unimportant.



Classification of state spill

Based on four common OS entity design patterns:

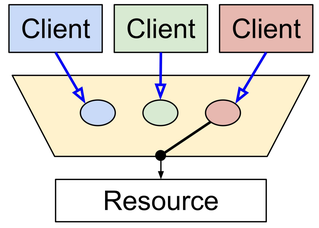


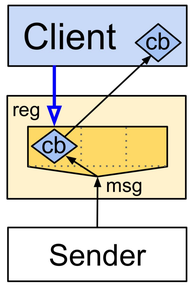
Indirection Layers convert between high-level and low-level representations of data and commands.

- Virtual File System abstraction
- Process abstraction
- Microkernel userspace servers
- Device drivers

Multiplexers temporally or spatially share an underlying resource among multiple clients.

- Schedulers / process mgmt
- Window managers
- High-level drivers



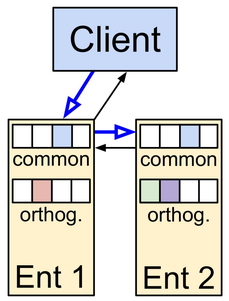


Dispatchers register client callbacks to properly deliver events or messages.

- Device event callbacks
- Synchronization primitives
- Upcalls
- IPC layers

Inter-Entity Collaboration requires synchronization of non-orthogonal states to ensure correctness.

- Microkernel userspace servers
- Android services

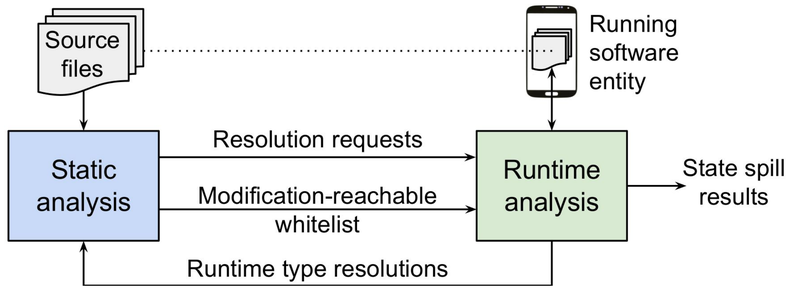


Designs to avoid state spill

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

RESTful principles

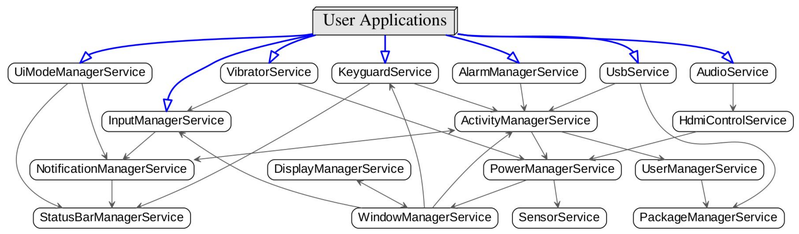
Automated detection with StateSpy



1) Detect *quiescent point* for safe analysis
   -- monitor transaction entry & exit points
2) Capture state of software entity
   -- key insight: use **debugging frameworks**
3) Difference captured states
   -- via existing tree comparison algorithms
4) Filter results with static analysis
   -- determine *modification reachability*

State spill in Android system services

- StateSpy found state spill in 94% of Android services analyzed, most with 1-10 instances
- Classified state spill instances in 60 transactions:
  - 39% caused by indirection layers
  - 21% caused by multiplexers
  - 55% from dispatchers/collaboration
- Better discovery of problems in app migration than manual identification of residual dependencies [1]
- Discovered *secondary spill* in 27 services:



[1] Alex Van't Hof, et al., *Flux: Multi-Surface Computing in Android*, EuroSys'15.