A Characterization of State Spill in Modern Operating Systems

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

Abstract

Understanding and managing the propagation of states in operating systems has become an intractable problem due to their sheer size and complexity. Despite modularization efforts, it remains a significant barrier to many contemporary computing goals: process migration, fault isolation and tolerance, live update, software virtualization, and more. Though many previous OS research endeavors have achieved these goals through ad-hoc, tedious methods, we argue that they have missed the underlying reason why these goals are so challenging: *state spill*.

State spill occurs when a software entity's state undergoes lasting changes as a result of a transaction from another entity. In order to increase awareness of state spill and its harmful effects, we conduct a thorough study of modern OSes and contribute a classification of design patterns that cause state spill. We present StateSpy, an automated tool that leverages cooperative static and runtime analysis to detect state spill in real software entities. Guided by StateSpy, we demonstrate the presence of state spill in 94% of Android system services. Finally, we analyze the harmful impacts of state spill and suggest alternative designs and strategies to mitigate them.

1. Introduction

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 [38, 55, 34]. 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 [28, 26, 52, 51]. 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 [7, 25, 27, 48, 5]. 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) [11, 6].

In this paper, 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, if an application interacts with a system service, causing that service to store application-relevant states in its memory, then state spill has occurred from the application to the service. We formally define state spill and the conditions under which it occurs in §3.

From this cursory explanation, it is apparent that state spill is a phenomenon that permeates all levels of a software system. However, it is relative to the choice of *entity granularity*: interactions that cross the boundary of an entity are considered transactions of interest, but those within a single entity are not. As an example, if the entity granularity is defined to be a process, then an IPC call between two processes could constitute state spill, but a local function call within a process could not. The definition of state spill is not confined to any particular entity granularity; entity granularity is a platform- and goal-specific decision (§3.1). However, in this work, our analysis centers around entity bounds akin to those of a module: an entity consists of a group of related functions and the data they modify. When analyzing Android system services, for example, the best entity granularity choice is a Java class.

The second contribution of this work is a manual analysis of state spill and its detrimental impact on the aforementioned computing goals across various real-world OSes. We find that

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4938-3/17/04. . . $15.00

DOI: http://dx.doi.org/10.1145/3064176.3064205

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 (§4) according to common design patterns: indirection layers, multiplexers, dispatchers, and inter-entity collaboration.

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. As described in §5, 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. The runtime analysis component of StateSpy captures a running entity's state by interposing on an entity's execution paths when handling transactions invoked by real-world applications. 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 (§6) 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.

We discuss how to mitigate the effects of state spill in §7 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. In fact, we have shown Android system services can be made more fault tolerant with state spill mitigation techniques [17].

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. StateSpy is open-source and available at [10].

2. Background and System Model

In order to properly define state spill and the conditions under which it occurs, we first assume a system model with a nested hierarchy of abstraction (Figure 1). Each layer of abstraction contains one or more *software entities*, a generic term that covers common programming and runtime abstractions, e.g., processes, threads, modules, classes, functions. The various granularities of software entities and how they pertain to state spill are described further in §3.1.

2.1 Transactions: Inter-Entity Communication

We model all communication between entities using the notion of *transactions*, which have IPC-like semantics.

Definition 1. A *transaction* is the flow of control (and optionally data) from one software entity to another. The *source* entity, or *client*, initiates the transaction and the *destination* entity, or *server*, receives and handles it, returning control to the source upon completion.

Transactions can represent procedure calls, system calls, interrupts, signals, other IPC, and more. The exact incarnation of a transaction is defined by which entities are involved; for example, a function call is a transaction from one function to another function in the same thread; system calls are transactions from a userspace entity to a kernelspace one; IPC is a transaction from one process to another. Transactions have the following explicitly-defined characteristics:

- If a transaction is interrupted (e.g., preempted), it is simply treated as incomplete until it resumes and returns control.
- If a transaction never completes, such as a call to an endlessly-looping function that listens for messages, no future actions from different source entities can be affected by its changes; thus, it is irrelevant to state spill.
- If a transaction handler fails, e.g., by returning an error, it is treated the same as a completed successful transaction.
- If a transaction is asynchronous/non-blocking, it is treated as two separate ones: an initial transaction from the source to the destination, and a second transaction from the destination back to the source as a completion event.

We are primarily interested in short-lived transactions that leave a lasting change on the destination entity, such that the latter's future behavior will be affected.

2.2 Quiescence in Software Entities

In assessing state spill, we are interested in the effect of a *single* transaction on an entity; thus an entity's state is only matters at certain points: before and after a transaction, not during. We use the concept of *quiescence* to identify this condition. Given how communication in our system model is entirely transaction-based, the quiescence of a software entity is determined solely by its transaction handling.

Definition 2. *Quiescence* is the stable condition of an entity when it has no in-progress, unfinished transactions from external entities. An entity is *dynamic* when not quiescent.

This differs from traditional definitions of quiescence, which typically specify that a given entity must be completely suspended, sleeping, or absent from all CPU runqueues [48, 53]. Other definitions of quiescence go even further by saying that an entity cannot be considered quiescent if any of its functions are on the call stack of any other process [5, 32, 25, 7]. Epoch-based quiescence utilizes an interposition layer to mediate access to a given entity, ensuring quiescence is reached once all other entities are finished interacting with the mediated entity [49, 27].

These interpretations of quiescence are far too strict for our needs, as they would cause nearly every entity to *never* reach quiescence, especially those that execute in the background or never terminate, e.g., system services and the kernel. While such definitions cause many event types to prevent quiescence, e.g., scheduler preemption, our definition simplifies it to occur *only* upon the completion of inter-entity transactions. This is necessary because, although an event like scheduler preemption may cause the entity to be non-runnable, that entity may still be in the midst of handling a lengthy transaction when it is blocked or otherwise taken off the runqueue.

2.3 State of a Software Entity

Informally, *state spill* occurs when a transaction leads to lasting changes in the destination entity. Before we formally define state spill (§3), we must first define what we consider to be part of a given software entity's state. As the term *software entity* covers a variety of possible incarnations (e.g., modules, processes, threads, classes), we consolidate its state definition into a reduced set of "core state" that all forms have in common, exclusive of process- or thread-specific states like the program counter, call stacks, and function frames.

Definition 3. The state of a software entity, denoted $\mathcal{E}$, consists of the set $\mathbb{V}$ of program variables within the entity's scope, and the values of those variables.

This definition confines entity state to include only the information within the entity's scope, i.e., everything visible within the entity's logical bounds. This is a fair, representative model of real-world OS entities who cannot control nor access external state due to permissions and secure information hiding policies. For example, the scope of a *module* includes the local variables in the functions within that module and the global variables that they access. The scope of a process entity is identical to the scope defined by the program executing within it, which includes all program variables but excludes the process control block and other OS state external to the process. For class object entities, all class member fields are considered to be in-scope throughout the entire duration of that class object, in addition to any local variables and public static members accessible from the current execution point.

Defining software entity state in this way necessarily enables us to treat varied OS components in a consistent way amenable to formalization, static analysis, and runtime analysis. Another benefit of this definition is its Markovianness, i.e., the exclusion of prior input, events, or control flow from an entity's state. This allows easy analysis of an entity: only current contents are inspected, prior condition is disregarded.

Quiescent Entity State

When an entity is quiescent, its state $\mathcal{E}^Q$ becomes a refinement of entity state $\mathcal{E}$ exclusive of temporary states. Formally, a *temporary state* is a program variable whose underlying lifetime [47] does not persist beyond the transaction currently being handled by the entity. For example, local variables with automatic lifetimes are temporary states, whereas class member fields or global variables in a process are non-temporary. Only lasting changes that stem from a transaction can present complex challenges to the aforementioned computing goals. Changes to temporary states are unimportant and irrelevant to state spill because they are entity-local and cannot affect the complex interdependencies and coupling between entities.

Definition 4. The state of a quiescent software entity $\mathcal{E}^Q$ consists of the set $\mathbb{V}^Q$ containing all *non-temporary* program variables within the entity's scope and their values.

Recall that our definition of quiescence allows entities to reach quiescence quite often — once after every transaction has completed given that other transactions have not yet begun — and thus enables us to isolate specific states changed by a single transaction. This, combined with the elimination of transient changes, leads to more accurate analyses with meaningful results: state spill that actually matters.

3. State Spill: A Formal Definition

We now formally define the concept of *state spill* and describe the conditions under which it occurs. When a source entity S initiates a transaction with a destination entity D, the state of D may undergo a lasting change while handling that transaction. State spill may be *explicit*, when data from S is passed to and stored in D, or *implicit*, in which no data is passed but handling the transaction incites a change in D on behalf of S.

Previous works have addressed problems related to state spill, such as data-flow analysis and taint tracking in systems software [59, 29, 60] and in Android [8, 18]. In general, these works focus on tracking the propagation of data throughout a system, often for privacy purposes, to determine what data can reach which entities. However, they are not concerned with the *changes* to other software entities and the subsequent effects that stem from said data propagation, which we have found is an often overlooked yet important occurrence. Awareness of data movement is a step in the right direction, but we argue that it is more useful to understand the *impact of that data movement* beyond privacy concerns (§8).

Formally, state spill occurs from $\mathtt{S} \multimap \mathtt{D}$ if there is a difference in entity D's quiescent state after handling a transaction from S, and the difference was caused by that transaction.

Definition 5. State spill $\mathtt{S} \multimap \mathtt{D}$ occurs if $\mathcal{E}^Q_{\mathtt{D}} \neq \mathcal{E}^{Q\prime}_{\mathtt{D}}$, in which

- $\mathcal{E}^Q_{\mathtt{D}}$: quiescent state of D *before* a transaction from S, and
- $\mathcal{E}^{Q\prime}_{\mathtt{D}}$: quiescent state of D *after* completing a transaction.

To determine whether $\mathcal{E}^Q_{\mathtt{D}} = \mathcal{E}^{Q\prime}_{\mathtt{D}}$, one must compare the non-temporary variables in entity D before and after a given transaction (Definition 4). Because entity quiescence ignores internal actions (Definition 2), one must also ignore all changes from those internal actions in $\mathcal{E}^Q_{\mathtt{D}}$ and $\mathcal{E}^{Q\prime}_{\mathtt{D}}$ and compare *only* states that can be modified by entity D's handling of that transaction.

Definition 6. Two quiescent entity states are equivalent if

$$\forall e \in \mathbb{V}^{Q\prime} \mid e \in \mathbb{M}_{\mathtt{t}},\ e \in \mathbb{V}^Q \ \wedge\ val^Q(e) = val^{Q\prime}(e),$$

in which e is a variable in the entity's post-transaction set of (non-temporary) variables, $\mathbb{M}_{\mathtt{t}}$ is the set of variables *actually modified* by the transaction handling, and $val(e)$ is the value of variable e. This equivalence is denoted $\mathcal{E}^Q = \mathcal{E}^{Q\prime}$.

In some systems, state equivalence testing may be impractical due to the size or highly dynamic nature of an entity's state. Although we have yet to encounter such an entity, an approximation of state equivalence testing may be more valuable than a direct comparison of entity state. For example, a good litmus test for state spill may be whether an entity S continues operating successfully after random changes are injected into entity D's state, without informing S of the change. We leave these imprecise approximations for future work.

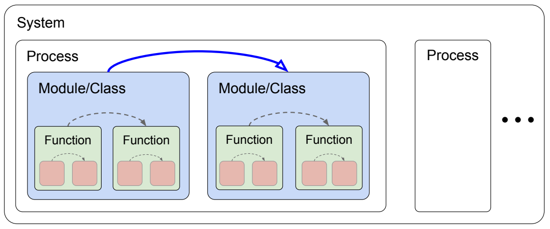

Figure 1: Entity granularities have a nested hierarchy; state spill is by definition relative to the chosen granularity. The module/class-based entity choice shown here allows state spill only from transactions between those entities (→); interactions between finer-grained entities (⇢) are irrelevant.

3.1 Relative Granularity of State Spill

Based on the above definitions, state spill is relative to both the spatial granularity of entities and the temporal grouping of transactions between those entities.

Spatial Granularity of Entities

Because transactions are by definition between two entities, state spill depends greatly upon the spatial granularity of the involved entities, i.e., how their bounds are defined. Entity granularities range from the highest level of considering the entire computer as a single entity to the lowest level of considering each individual function or even basic block as a fine-grained entity. At the highest granularity level of an entire computer being a single entity, state spill occurs from one computer to another via over-the-network transactions; any interaction between applications within a computer is irrelevant. Moving down a level, if an entire multi-process application was considered a single entity, then state spill happens from one application to another due to inter-application transactions; any interactions within that application's processes or threads are irrelevant. At the lowest level, in which each function is considered its own fine-grained entity, transactions between functions (function calls) across the entire system are eligible to cause state spill, making state spill ubiquitous and uninteresting. Thus, both very high and very low levels of entity granularity are neither useful nor appropriate for analysis, not to mention impractical.

The choice of entity granularity for state spill analysis is both platform-specific and goal-specific. For example, if one wishes to determine the barriers to process migration in a Unix-like system, each process should be its own entity such that state spill would be detected in transactions beyond a process's address space. If one wishes to live update a single function in isolation without having to worry about its dependencies on other functions, entity granularity should be set to a single function in order to track inter-function state spill. For analysis of Android system services (§6), a class-based entity granularity is more appropriate because each service is implemented as an OOP class.

In this work, we take the middle ground by choosing a moderately coarse entity granularity. The coarsest granularity of an entire computer as one entity has already been studied and addressed in the form of RESTful software architectures [23] used across web services [39]. In theory, RESTful design principles ensure that transactions between client and

```
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++;
  }
```

Listing 1: System service implemented as an OOP class.

server systems are self-contained and require no pre-existing state to be held on the server, preventing state spill from the outset. We choose a finer granularity than whole-computer entities because state spill in mid-level entities is more relevant to the systems research goals we target. Moreover, this choice produces a balance of meaningful results, whereas granularities that are overly fine or coarse are susceptible to state spill in *all* interactions or *no* interactions, respectively.

Temporal Granularity of Transactions

In addition to the spatial dimension of entity granularity, state spill also has a temporal dimension: the granularity of a transaction. Temporal transaction granularity specifies how many consecutive interactions between two given entities are grouped together into a single logical transaction. The exact form of a transaction is orthogonal to its temporal granularity and dictated solely by entity granularity, e.g., a transaction between two function entities *must* be a function call.

While this work focuses on analyzing one transaction at a time, it can be useful to collapse multiple transactions into one. When analyzing state spill in a common procedure of three transactions in series, e.g., connecting to and configuring updates from a sensor service, there is no point determining which states are spilled during the first two intermediate transactions. In fact, doing so during a tool-based analysis would create unnecessary overhead, especially considering that state spill only *needs* to be analyzed with respect to two quiescent points (Definition 5), not an individual transaction. Thus, a transaction's temporal granularity is another dimension to consider when understanding and detecting state spill.

3.2 Simple Example of State Spill

We use the sample system service of Listing 1 to illustrate specifically how applications might spill state into system service entities. In this example, the service is implemented in an object-oriented language; thus, its entity granularity is a *class*, its bounds include all class member fields, and its public methods are transaction handlers/entry points. Such services are typically instantiated once upon boot and never terminate, with much longer lifetimes than applications.

When an application initiates an `addCallback` transaction, both the `cf` and `cb` parameters cause explicit state spill by changing `mConfig` and `mCallbacks` respectively, whose lifetimes persist beyond that transaction. In contrast, although the `id` parameter is passed into `addCallback`, it

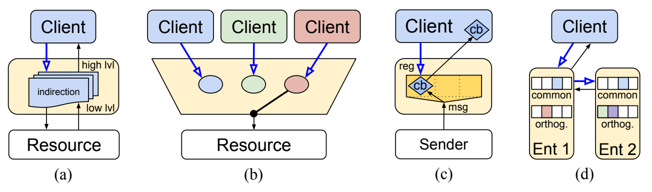

Figure 2: Common design patterns that cause state spill ($-\!\!\triangleright$). (a) Indirection Layers cause state spill when converting between two representations of data/commands; (b) Multiplexers harbor state spill when serving multiple clients; (c) Dispatchers harbor state spill by holding registered callbacks (`cb`) for message/event delivery. (d) Inter-entity collaboration causes state spill when non-orthogonal states must be synchronized.

does not constitute state spill because its lifetime is temporary (function-local) and ends when the transaction returns. The `sCount` field has a lifetime tied to the lifetime of the underlying runtime (JVM) instance, potentially longer than that of the `SystemService` entity; `sCount` is an example of implicit state spill because its value changes as a result of the transaction despite not being directly passed from the application. Finally, the `unrelated` field is not actually modified by the `addCallback` transaction, ($\mathtt{unrelated} \notin \mathbb{M}_{\mathtt{t}}$); thus, although its value may have been changed by another thread during the transaction, it is excluded from quiescent state equivalence testing and cannot contribute to state spill.

Formally, the state of the `SystemService` entity `D` at several execution points in time is as follows:

1. Upon service boot but before any transaction occurs, `D` is initially quiescent, thus $\mathcal{E}_{\mathtt{D}}^{Q}$ consists of $\mathbb{V}_{\mathtt{D}} = \{\mathtt{sCount}, \mathtt{mConfig}, \mathtt{mCallbacks}\}$ with values $\langle 0, 0, \{\varnothing\}\rangle$.
2. `D` becomes dynamic in L9, and by L10 has additional temporary variables $\{\mathtt{id}, \mathtt{cf}, \mathtt{cb}, \mathtt{b}\}$.
3. After L13, `D` returns to quiescence, thus $\mathcal{E}_{\mathtt{D}}^{Q\prime}$ contains $\mathbb{V}_{\mathtt{D}}^{Q\prime} = \{\mathtt{sCount}, \mathtt{mConfig}, \mathtt{mCallbacks}\}$, whose corresponding variable contents are $\langle 1, \mathtt{cf}, \{\mathtt{cb}\}\rangle$.

Therefore, because $\mathcal{E}_{\mathtt{D}}^{Q\prime} \neq \mathcal{E}_{\mathtt{D}}^{Q}$, Definition 5 stipulates that the `addCallback` transaction causes state spill from $\mathtt{S} -\!\!\triangleright \mathtt{D}$.

4. Classification of State Spill and its Harmful Effects

We now take a broad look at various real-world operating systems to demonstrate the ubiquity and harmful effects (**boldfaced**) of state spill therein. In doing so, we contribute a classification of state spill based on various entity design patterns: *indirection layers*, *multiplexers*, *dispatchers*, and *inter-entity collaboration*. Although other forms exist, these concisely represent the vast majority of state spill in OS entities and their roles in causing it. In §7, we discuss spill-free alternatives to these designs.

4.1 State Spill in Indirection Layers

The first form of state spill, depicted in Figure 2(a), occurs when a client interacts with an *indirection layer* to access a lower-level resource. Indirection Layers can be any horizontal slice in a multi-layered software stack: for example, an application (client) can use APIs provided by a library (indirection layer); a system service (client) can access I/O devices through the kernel driver (indirection layer). The indirection layer translates client requests expressed at a high level of abstraction into lower-level commands that the underlying resource can understand; in doing so, the indirection layer harbors spilled state from the client entity, e.g., information about how the client is utilizing the resource. Developers of indirection layer entities necessarily choose to store client-specific information internally to (*i*) ensure that access privileges are upheld, and (*ii*) maintain a notion of client progress during a series of client-resource translations.

Common Abstractions: Processes and VFS

The process abstraction is an indirection layer that allows unprivileged user programs to safely access the CPU without understanding the underlying hardware. Most monolithic OSes implement the process abstraction via an indirection layer in the kernel and maintain per-process metadata as a single list therein, e.g., Linux kernel's `task_struct` list. The user program running atop and utilizing the process abstraction will cause state spill beyond its bounds into `task_struct`'s member variables and others within the kernel's process management subsystem.

This state spill stems from a design choice to have all process-related information in a convenient centralized place, and also from the need to protect such information from malicious userspace tampering. However, it renders **process migration** infeasible because one must track and retrieve states spilled from a process into other OS entities (the kernel and system daemons), a complex and impossible task. This is a recognized problem previously termed "residual dependencies" [34, 55], but is a symptom of state spill. As discussed in the next section, microkernel OSes also harbor similar state spill, but within userspace servers that implement indirection layers rather than the kernel.

The Virtual File System (VFS) indirection layer, present in monolithic and microkernel OSes alike, provides a simple file abstraction for user processes (clients) to access low-level device drivers (resources). We take Linux's VFS implementation and I^2C driver as an example. The VFS entity is itself an indirection layer, whose state includes a `struct file` that manages and holds references to the underlying I^2C device and driver as part of its `private_data` element. When a client process issues an `ioctl` transaction on that VFS entity's *device file*, the kernel routes the operation from the VFS layer to the corresponding driver's `ioctl` implementation, `i2cdev_ioctl` in the case of I^2C. Because it was initiated by VFS, this driver holds a reference to the VFS's I^2C `file` structure and its functions can directly modify VFS states, e.g., file mode, readahead, mutex, ownership permissions; other operations like `read` and `write` behave similarly and modify other VFS entity states, e.g., its `file_pos` offset.

This is a deceptive form of implicit state spill: the VFS layer *appears* to not modify its own state or keep data passed in from userspace, but rather its state is modified transparently by the driver entity hidden beneath the abstraction provider itself. This design choice was made for convenience and efficiency reasons: since the VFS and driver entities share a single kernel address space, it is easier for developers and faster to directly update the elements in the VFS's `file` structure using shared references rather than explicit message passing. Similar state spill exists in the userspace VFS servers of microkernel OSes like MINIX 3, Genode, and others.

Microkernel Userspace Servers

Microkernel-based OSes like MINIX and seL4 move the vast majority of OS functionality into userspace servers in favor of a very small kernel core. These servers are often indirection layers that act as middlemen between applications and the microkernel, e.g., converting POSIX API calls into MINIX-specific functions that their microkernel can handle. This software architecture results in state spill that directly impedes **live update** and **hot-swapping** of microkernel servers, as evidenced by the MINIX authors' tedious, ad-hoc undertaking to enable live updates in MINIX 3 [25].

Besides abstraction levels, userspace servers also bridge privilege levels: userspace servers have more privileges than applications but less than the kernel, e.g., they cannot do context switches or top-half interrupt handlers. The introduction of different privilege and abstraction levels is a design choice that prioritizes modularity and the minimization of kernel size at the cost of high susceptibility to state spill. For example, MINIX 3's userspace scheduler SCHED [50] is an indirection layer that sits between user processes and the microkernel's context switch *mechanism* to control the system's scheduling *policy*. SCHED lacks context switch privileges, so it simply chooses the next target process to be run and relies upon the kernel's context switching mechanism (`sys_schedctl`). The `sys_schedctl` system call copies process-relevant parameters into the kernel's list of `struct procs` — a prime example of state spill — before actually triggering the context switch. Spilled states include the target process's scheduling flags, endpoint, parent endpoint, priority, timeslice quantum, CPU affinity mask, schedule-enqueue bit, and more.

The same userspace-policy kernelspace-mechanism structure is used for many other MINIX 3 servers, such as the process manager (§4.2), virtual file system, memory manager, and reincarnation server, all of which cause similar forms of state spill. We omit their details for brevity's sake, but such state spill is a direct obstacle to realizing **live update**, **hot-swapping**, and **virtualization** of said userspace servers.

4.2 State Spill when Multiplexing

The second form of state spill commonly occurs when an entity acts as a *multiplexer* that allows multiple clients to access a single underlying resource by means of sharing it temporally or spatially. As depicted in Figure 2(b), a multiplexer entity causes state spill by harboring states that correspond to each client's individual interactions with the resource. This form of state spill typically occurs out of necessity, in that the multiplexing entity must maintain contextual data about its users in order to properly partition and manage its resource. For example, a device driver holds a representation of each process's usage of its hardware peripheral; the virtual memory subsystem contains mappings and other allocation details to multiplex applications' accesses to physical memory.

Process Management

Process Management (PM) entities temporally multiplex access to the underlying CPU (resource) among multiple user processes (clients). They maintain metadata about every client process in order to monitor and control them, e.g., deciding when and what to run. This process-specific metadata is a prime example of state spill that exists across many OS designs and negatively impacts the computing goals below.

For example, when one process creates another via `fork`, the PM multiplexer creates a set of data structures to represent that new process's state, a form of implicit state spill. In monolithic OSes like Linux, this spill occurs from a process to the kernel's PM subsystem; in microkernel OSes like MINIX 3, this spill occurs from a process to the userspace PM server. In addition to process creation, other PM actions cause state spill; as processes execute and interact with peripherals and other entities, the PM entities in userspace servers or in the kernel must update their *process tables* accordingly to reflect the process's new condition. For example, when a process accesses an I/O device by invoking the driver's `read()` system call, the device driver may block that process by setting a flag in its process table entry while fetching the requested data, causing convoluted state spill from the client process to the PM multiplexer via the driver.

Not only does state spill in multiplexers induce the aforementioned residual dependency problem that hampers process migration (§4.1), but it also breaks **fault isolation** guarantees by inextricably tying the states of its client entities together in a single entity, causing fate sharing. That is, if one process causes corruption or faults in the PM server, other processes will also be affected. On a related note, **fault tolerance** in multiplexers is impossible in the face of state spill: restoring a failed server instance on behalf of one client process may be successful, but it will fatally disrupt any other clients using that server due to the unexpected absence or change of server-side states [11]. In fact, this is true for most indirection layers and multiplexers, not just PM entities.

The architectural philosophy of monolithic and microkernel OSes may necessitate such state spill, but some experimental OSes reduce it with unique approaches. Genode [1] uses a hierarchical PM technique in which an entity that creates a process maintains metadata and control over that new process. This does reduce the extent of state spill into PM entities, but does not fully decouple the creator or its created process from said PM entities.

Window Management

While the above PM multiplexers are *temporal*, window management (WM) systems are *spatial* multiplexers of graphical resources like framebuffers. Client applications create and manipulate their view contents by submitting requests that contain application-specific state to the WM multiplexer, which assigns a region of the screen/framebuffer resource to that application. The state stored in the WM entity when handling an application's request constitutes state spill from application to WM entity. WM entities also harbor explicit state spill when receiving configuration data and window content from the applications, which they store directly instead of allowing applications control over their own data.

The X window system [45], though designed as loosely-coupled modules, harbors such state spill among various multiplexer components. One example is the X server, an entity that multiplexes the local keyboard, mouse, and display resources between many client applications, storing client-specific information in the server entity when a given client requests window creation or content display. Other actions handled by the X server include centralized reparenting of windows and caching of offscreen graphics (for context menus and transient pop-up windows) to avoid client-server round trips; these actions require states from application requests to be kept in the X server entity's local storage, necessarily causing state spill from applications into the X server multiplexer. Another X component that causes state spill is its window manager, e.g., Compiz, KWin, a multiplexer that maintains Z-depth and other layout data on behalf of each application window in order to properly position windows relative to one another.

Nitpicker, the atypical WM multiplexer in Genode [21], forces each client to take ownership of its private views and buffers in an effort to improve inter-client security. Nitpicker then accesses these client buffers via shared memory mappings, only upon an explicit client request. This reduces state spill by allowing clients to allocate and manage their own view buffers, but still harbors some spilled states in the form per-client metadata: depth layering information, thumbnails of each view, keyboard shortcuts, etc.

In WM multiplexers, as with PM multiplexers above, state spill violates **fault isolation**, which makes **hot-swapping** and **live update** exceedingly difficult because the separate entities (e.g., client and X server) cannot be updated in isolation or swapped independently. Updating multiple separate entities atomically is a requirement for consistency — many live update works [7, 25, 27, 48, 5] devote the bulk of their efforts to determining quiescence and accommodating states distributed across multiple entities — but is practically infeasible for windowing systems with many tightly-coupled components. State spill in WMs does not significantly complicate process migration because a WM entity will likely need to reconstruct an application's graphical window and displayed content on the new target machine post-migration. For **security**, state spill in WM multiplexers is particularly harmful because private content may be revealed to other client windows via the shared multiplexer medium. Due to spilled states stored communally in the multiplexer entity, malicious X client applications may be capable of tampering with other windows, injecting false keystrokes or commandeering them for keylogging purposes, or stealing the contents of shared buffers (e.g., clipboard data), and more [4]. A crafted request from an X client can even cause the X server to overwrite arbitrary memory of window buffers spilled into its multiplexer entity [4], causing arbitrary code execution or a denial of service for clients reliant upon those buffers.

4.3 State Spill in in Dispatchers

The third form of state spill arises in *dispatchers* that allow client entities to register callbacks in order to receive events or messages from a sending entity, as depicted in Figure 2(c). For example, an application that needs to wait for a particular event may register a callback with the daemon or kernel that receives the event; this is very common in event-driven programming models. Dispatchers necessarily cause state spill by holding client-provided callback references in order to properly route communication between entities. In fact, this frequently appears as a subcomponent of other patterns; for example, the X server (client) registers callbacks in the kernel's I/O driver (multiplexer) to receive HID events.

In many IPC implementations, dispatchers cause state spill; for example, System V IPC dispatcher entities in the kernel maintain `ipc_perm` keys mapped to external reference IDs and synchronize `msgque` vectors and `task_struct` IPC status bytes. Unix domain sockets, inotify, and d-bus subsystems are similar. Although signals do not necessarily require callback registration, the Linux kernel's signal dispatcher can cause state spill when, for example, the signal `blocked` mask or `sigaction` array is changed in a given process's task descriptor, or a signaled process is blocked. Finally, mutexes and locking features can also contribute to state spill; for example, the semaphore implementation in MINIX 3 is a dispatcher entity that implements mutex by adding references to the waiting processes in the PM multiplexer's process table, such that it can dispatch mutex release events to those waiting processes. These references, although necessary, do constitute state spill when a process utilizes semaphores.

In the Swift system [14], a lower-layer entity (sender) is able to invoke a procedure in a higher-layer entity (client) using the *upcall* mechanism offered by an intermediate entity (dispatcher). State spill occurs because the higher-layer entity must *downcall* into the dispatcher to register itself such that future upcalls can deliver the correct execution context for the higher-layer entity. Furthermore, because key procedures in the lower layers of the Swift system rely on the correct implementation of higher-layer procedures, à la polymorphism, those procedures tend to access common data across a "vertical stripe" spanning several layers.

The author of Swift recognized this issue and attempted to rectify it by mandating that all client-facing common data be unlocked and consistent before an upcall; however, this guarantee was difficult to implement. Any data in this vertical stripe *not* protected by mutexes contributes to state spill, resulting in interdependent coupling between layered entities, a direct challenge to **fault isolation**, **fault tolerance**, and **maintainability**. In fact, this problem and other symptoms of state spill in upcall mechanisms, such as unpredictable control flow hazards, are barriers to proper **security**; hence, dispatchers supporting arbitrary upcalls to userspace were never accepted into the mainline Linux kernel [9, 2].

4.4 State Spill due to Inter-Entity Collaboration

Finally, the fourth form of state spill occurs when multiple entities communicate with each other to ensure correctness and consistency in their view of the OS, as depicted in Figure 2(d). This stems from a desire to accomplish separation of concerns, in which a complex OS feature like process creation is broken up into a series of smaller duties that are each assigned to specific entities (usually system services or daemon processes). On paper, each entity's responsibilities and states are orthogonal, leading to a more modular software architec-

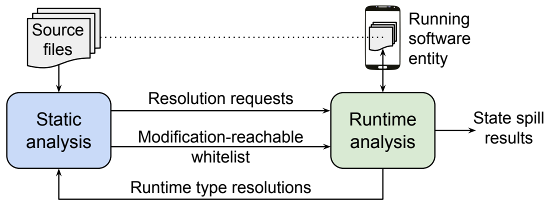

Figure 3: StateSpy employs both runtime and static analysis in a cooperative feedback loop to accurately detect state spill.

ture; however, in reality, each entity has *common* states that are either shared with or dependent upon other entities. The common information — not necessarily identical replicas — must therefore be synchronized, causing state spill that harms **maintainability**, **live update**, and more.

For example, MINIX 3 divides such complex OS tasks among multiple userspace servers, each of which maintains its own version of key data structures, e.g., process tables, to fulfill its duties. When an application on MINIX 3 wishes to set its `uid` or `gid`, the PM server is the first to receive and handle that call. However, the PM's duties do not extend to the filesystem, so it must ask the VFS server to complete the file-related aspects of the call, and then proceed to collaborating with the memory management (MM) server, SCHED, and so on; each server has its own version of common data structures that must be synchronized. This pattern occurs in many other POSIX calls, e.g., exec, fork, virtual memory functions, and also in other microkernel OSes and Android. This form of state spill is particularly insidious because while the user may anticipate state spill in the transaction target (the PM server), the synchronization-induced state spill in other entities (VFS, scheduler, MM) is completely obscured.

5. StateSpy: Automated State Spill Analysis

In order to detect and analyze state spill in real-world systems, we design and implement StateSpy, a tool suite that automates the discovery of state spill and assists developers in understanding the conditions under which state spill can occur in their entities. Developers simply connect StateSpy to an existing entity running in a real OS and allow it to execute as normal (ideally with a variety of inputs), after which results are automatically outputted containing all detected state spill occurrences and the actions that caused them. StateSpy also accepts as input the source code of an entity to help limit the scope of analysis and prune false results.

As shown in Figure 3, StateSpy's runtime analysis component works cooperatively with the static analysis component to generate state spill results. Our approach is related to concolic testing [46] in that it blends symbolic execution-like static analysis with runtime analysis, but inverts concolic testing because our results come directly from the runtime analysis component. As concolic testing begins from a single real execution trace, it may inadvertently eliminate some potential execution paths based on those initial runtime seed values, causing poor coverage. However, because StateSpy's static analysis conservatively eliminates irrelevant states and not paths, its runtime results cannot miss state spill instances in any Java code path.

The following sections describe the challenges in designing said joint analysis and the realization of StateSpy itself. We implemented StateSpy's runtime analysis and static analysis in 1643 and 2648 lines of Java code, respectively. We strive to keep our design OS-agnostic wherever possible; however, as StateSpy currently targets Android system services (§6), some aspects are Java-specific. A tool for native, non-managed systems languages (e.g., C/C++) is easily realizable using techniques similar to those described below.

5.1 Challenges of State Spill Analysis

In designing a state spill analysis tool, one must address four main challenges: detecting quiescence in an entity, capturing an entity's states *with meaningful context*, differencing those states, and filtering out irrelevant results. The first three challenges are addressed by the runtime analysis component of StateSpy, the last by static analysis. An additional challenge stems from the shortcomings inherent in runtime analysis and static analysis when trying to jointly use both techniques; this is addressed in §5.4.

Detecting Quiescence

In the general case, it is difficult to determine when an entity is quiescent because quiescence is often specified via environment-specific conditions. However, our *transaction-based* definition of quiescence simplifies this: we only need to detect transactions incoming to an entity and when those handlers have completed. In Android, this is relatively straightforward because transactions between entities have a clear entry and exit point, due to the strict nature of the Binder IPC protocol. Thus, StateSpy can simply monitor the entity's execution to pause threads at the beginning of a specified Binder method (e.g., `onTransact`), creating a quiescent period. In other systems, StateSpy can accept developer-defined entry and exit points to determine what constitutes a transaction.

Capturing a Software Entity's State

Capturing the state of an arbitrary software entity remains a hard problem that revolves around a tradeoff between genericness and accuracy. State capture approaches fall under two broad headings: (*i*) capturing an entity's underlying memory contents beneath the runtime environment, or (*ii*) capturing its state contents at the language level.

The former approach (*i*) is generic — no language-level support, understanding, or modification is necessary — but requires bridging the gap to recover the semantic knowledge of those captured memory contents, which remains an open problem in the forensic science domain. For example, if the kernel transparently captured a user process's address space, it would not know which memory contents represented which states. Thus, although this approach could support arbitrary software entities, classifying entity states (or their underlying memory) as temporary, modification-reachable, or any other trait would be infeasible, preventing us from determining whether they contribute to state spill.

The latter approach (*ii*) is less generic — it only works for a given language or runtime environment — but preserves contextual metadata like variable descriptors and type information. Meaningful context is vital to understanding and classifying states in an entity to accurately detect state spill.

Therefore, we adopt a language-level approach that avoids the gap between state values and their semantic meanings, allowing STATESPY to preserve states in their original form for analysis. The challenges in designing language-level state capture approaches are best evidenced by the following shortcomings of existing approaches.

- **Static analysis** cannot always deterministically guarantee whether a given state *will* change, only that it *may* potentially change. In addition, its inability to accurately resolve abstract types or methods prevents the full traversal of all possible execution paths.
- **Record and replay** is generally easier to implement than checkpoint-based state capture, but cannot inspect the *actual contents* of changed states, just the actions that evoked those changes.
- **Serialization** requires special support from the language on a per-type basis, which most legacy systems do not offer, and is infeasible to implement generically.
- **Runtime instrumentation** requires modifying the runtime to expose state information, an incredibly complex approach that risks disrupting the entity's behavior or even violating its correctness. Also, any executable code that is compiled into native or machine binaries will bypass the runtime, making it impossible to interpose upon that code even with the proper hooks available. Although runtime plug-ins allow for this unreliable introspection, many runtime implementations lack support for standardized features and hooks that these plug-ins rely on, e.g., Android's ART/Dalvik runtime.

To overcome these challenges, our key insight is to leverage *debugging frameworks* that already exist in the runtime or execution environment in order to non-invasively capture entity state (§5.2). Exploiting debugging extensions is a flexible technique generalizable to nearly all other platforms and systems, providing access to entity states with the full contextual information necessary for state spill analysis.

Differencing Captured States

In order to determine whether a transaction has resulted in state spill, one must difference an entity's quiescent state before and after that transaction. This is difficult because each state must be compared according to its underlying details, e.g., variable type and size. Essentially, this requires a full semantic understanding of each state, because comparing a list of integers is different than comparing two custom `structs`. Fortunately, the correct differencing of states goes hand-in-hand with the proper capture of states above; that is, utilizing debugging frameworks also provides sufficient state metadata for STATESPY to conduct proper state comparison.

In addition to the challenge of correct semantic state comparison, one must address the challenge of representing captured states in a way that supports arbitrary structure, circular references, and hierarchical relationships among states. For this, STATESPY builds a tree-like cyclical digraph that mirrors the structure of object states in the running entity (§5.2). With a proper structural representation and semantic understanding of entity states, we can apply existing tree comparison algorithms to identify which states changed during a transaction, i.e., occurrences of state spill.

Filtering Results

Delivering only relevant state spill occurrences is difficult because there is no ground truth for what actually constitutes state spill, aside from an expert developer's determination. Runtime analysis cannot differentiate between states that were changed as a direct result of the transaction and states that happened to change during the transaction (e.g., by background threads), a condition we term *modification reachability*, derived from Definition 5. This leads to a potential abundance of false positive results, i.e., when a state changed during a transaction but not due to that transaction. To remedy this, we rely on static analysis to assess the modification reachability of all states in an entity, described in §5.3.

5.2 Runtime Analysis Design

STATESPY's runtime analysis component detects state spill according to Definition 5: an entity's state is captured twice, at quiescent points before and after a transaction, and then differenced to identify spilled states. To address the first three challenges above, we leverage language-level debugging frameworks that can determine if an entity is quiescent, non-intrusively access its state, and obtain full contextual metadata to gain a semantic understanding of all entity states. Despite inherent language-specificness, we avoid environment-specific features — watchpoints are unsupported on many JVMs like ART/Dalvik — and strive to keep our key design concepts language-agnostic, such that the core ideas are portable even though the implementation is not. Utilizing debugging frameworks does indeed have many advantages:

- All managed language runtimes offer robust debugging extensions, even those that do not support instrumentation or specialized plug-ins;
- Introspection via debugging hooks is risk-free and cannot compromise the correctness of the entity or runtime;
- Debugging support is also available in execution environments without an underlying runtime or virtualization layer, such as native C/C++ processes.

The runtime analysis component of STATESPY is designed as a standalone application that runs in a separate *host* runtime from the *target* runtime, which contains the entity under analysis. It builds upon standard debugger hooks, e.g., breakpoints, variable inspection, expression evaluation, in order to pause the target runtime and induce quiescence before accessing the state contents within. In our Java-specific implementation of STATESPY's runtime analysis, the software entity of interest runs in the target JVM runtime and consists of a Java *object*, an instance of a Java class. That class contains transaction-handling methods that modify the object's states, i.e., its member fields.

The full procedure undertaken by the runtime tool is:

1. Attach to the target JVM process, and wait for quiescence by establishing breakpoints at the entry of every transaction handler method in the entity object.
2. When an entry breakpoint is hit, induce quiescence by suspending threads in the target JVM. Then, capture the full state of the entity object into the host JVM as $\mathcal{E}^Q$.
3. Set breakpoints at the exit points of the current transaction handler method and resume threads in the target JVM.
4. When the exit breakpoint is hit, induce quiescence yet again by suspending the target JVM's threads. Capture the full state of the entity object as $\mathcal{E}^{Q\prime}$.
5. Disable the exit breakpoints, re-enable all entry breakpoints from Step 1, and resume threads in the target JVM.
6. Finally, in the host JVM, compute the difference in pre- and post-transaction entity state, $\mathcal{E}^{Q\prime} - \mathcal{E}^Q$, which represents the state spill caused by that transaction.

However, even with debugging frameworks available, capturing the full state of an entity class object is surprisingly non-trivial. We cannot simply create a shallow copy or duplicate the references to the object's state inside the target JVM, for two reasons: (*i*) it would cause the pre-transaction state values to be overwritten by the post-transaction values, as the two objects are one in the same and cannot co-exist; (*ii*) it violates our guarantee that we will not intrusively modify the target JVM's internal states.

Instead, StateSpy must fully explore the entity's complex object graph, starting at the top-level class definition and recursively following all of its field references to other subclass instances. This depth-first exploration continues until a bottom-level primitive object is reached, at which point a custom *tree-based representation* of the object graph is generated in the tool's memory (host JVM) that mirrors both the object references and primitive values of the entity in the target JVM. This tree representation bears several advantages: it matches developers' intuition and is amenable to existing differencing algorithms and visualization tools.

To reduce execution time and storage space, StateSpy cherry picks which object references or primitive values to include or exclude. This is accomplished through the modification reachability whitelist from the static analysis component, but can also be manually adjusted by developers. In addition, StateSpy is aware of an object's type, semantic value, and any references to it; this enables construction of trees with uniquely-identifiable nodes, supporting hierarchical containment and circular references while avoiding the redundant capture of already-encountered nodes.

5.3 Static Analysis Design

As previously mentioned, the primary shortcoming of runtime analysis above is its inability to distinguish between states that changed during a transaction and states that changed *as a direct result* of that transaction, leading to potential false positives. Therefore, the sole objective of our static analysis is to address that ambiguity, i.e., to solve the *modification reachability* problem. StateSpy's static analysis produces a conservative whitelist including only the states that are or may be modification reachable. This problem is reminiscent of taint tracking and data-flow analysis (see §8), but handles the additional complex case of implicit state spill that is undetectable by information flow analysis and better addressed with techniques like symbolic execution. However, data-flow's requirements for source/sink identification and symbolic execution's term-based representation of object values are both inappropriate for determining modification reachability, so we develop our own algorithm.

StateSpy's static analysis algorithm recursively explores every instruction of every method reachable from the entry point of each transaction handler method in the target entity. Starting with an initial set of *variables of interest* (VOI), containing the target entity's non-constant member fields, the algorithm propagates VOI and variables that have been modified or aliased to and from every method invocation. We use the Soot framework [54] to analyze Jimple, a simple intermediate representation of Java, which means that variables can only be modified when on the left side of an assignment statement, and VOI can only be aliased when on the left side of an assignment statement as well.

Effectively, this technique is a form of forward program slicing [57] that selects and analyzes only the instructions in that slice, i.e., those capable of modifying any VOI. However, forward slicing suffers from search space explosion; to mitigate this, we only track and propagate the modification or aliasing of VOI, not the propagation of *all* variables. To further reduce analysis time, we cache which variables are modified and aliased by each method; for the sake of reusability, these are expressed as string literals (e.g., `this`, `return`, `param#`) instead of actual variable identifiers. Then, the next time a cached method is encountered, we instantly know whether the base object, return value, or parameter values will be modified or aliased by that method.

The full algorithm, given in [10], outputs a per-transaction list of modification-reachable fields in the target entity, which is fed into the runtime analysis tool to reduce false positives.

5.4 Bridging the Gap with a Feedback Loop

As previously mentioned, we establish a feedback loop (Figure 3) between the runtime and static analysis components of StateSpy to address the shortcomings of each. However, this introduces a circular dependency: static analysis alone is unable to fully explore all methods due to ambiguous declared types or multiple candidate implementations for a given method invocation, so it needs type resolution information from the runtime component to accurate resolve these ambiguities; runtime analysis needs resolution requests and whitelists from the static component. However, runtime analysis can only resolve types it encounters in real execution, but does not know which execution paths to explore until the static component requests mappings for missing types.

To address this catch 22, we break the circular dependency by synthesizing artificial test cases that force the runtime tool to traverse an execution path containing the runtime type resolution of the abstract declared type in question, which is then fed back into the static analysis component. We effectively bootstrap the static analysis tool with a few type resolution mappings and the runtime tool with a simple all-included whitelist, and then iteratively increase the type mappings set while reducing the whitelist according to modification reachability. Since all feedback data passed between the components does not change across analysis runs, we save this information persistently for future usage.

5.5 Validating StateSpy's Limitations

Currently, our StateSpy implementation strives for completeness over soundness, but guarantees neither. For clarity, StateSpy is considered *complete* if it does not omit any results that are actually state spill (i.e., no false negatives), and *sound* if all results it returns are indeed true state spill instances (i.e., no false positives). We aim to eliminate false negatives while still minimizing false positives. In this case, a false positive is a changed state identified as state spill by our tool that is actually not true state spill, whereas a false negative is a real case of state spill that our tool missed.

We applied StateSpy to a random sampling of 60 transactions in Android system services and manually verified the results. We found that StateSpy achieves a false positive rate under 11% across these 60 transactions, most of which stems from our immature support for native methods. Without static analysis, the runtime analysis alone results in a false positive rate well over 60% for some services, highlighting the necessity of the modification reachability constraint. Due to the sheer magnitude of transactions in Android and the lack of ground truth, we are unable to accurately assess StateSpy's false negative rate; however, a case study in §6.4 shows false negatives can occasionally occur.

As mentioned above, we conservatively include a state in the whitelist if static analysis cannot determine whether it is modification reachable, due to reasons like dynamic dispatch, unfollowable native code, or opaque external type references, Based on our experience, we believe that from a developer's perspective, determining the legitimacy of a state spill result is relatively easy, whereas discovering state spill results that were mistakenly omitted by StateSpy (false negatives) is practically impossible. Thus, false positives are less harmful than false negatives, so we prioritize completeness.

6. State Spill in Android

While §4 studied and classified state spill in a broad variety of OS entities, this section takes a deep dive into Android system services guided by StateSpy. Our results show that state spill is a complex, widespread problem that permeates the vast majority of Android system services and is detrimental to the aforementioned computing goals.

6.1 Experimental Methodology

Thus far, we have presented examples of state spill on a per-entity basis, but we now describe state spill on a per-transaction basis because StateSpy is capable of monitoring individual transactions. Transaction-oriented results are more useful to developers looking to isolate causes of state spill in their entities; knowing that a given entity harbors state spill is less precise than knowing which transaction caused it.

To obtain a large, representative body of transactions for StateSpy to analyze, we downloaded 150 Android applications from various top categories in the Google Play store and utilized the `monkey` [3] UI automation tool to run each application with hundreds of random yet meaningful inputs. It is unimportant which applications are run, only that they invoke a large body of transactions across many system services, which StateSpy can automatically monitor for state spill analysis. We believe that analyzing a varied, real-world set of transactions is preferable to a generated set of contrived test cases that may be unrealistic.

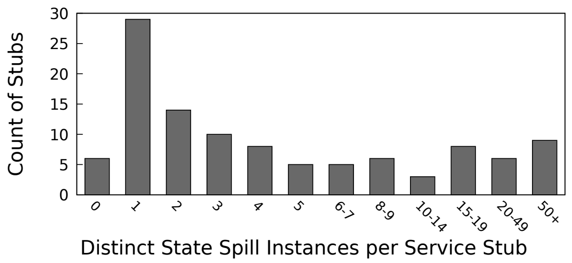

Figure 4: A histogram of how many distinct state spill instances occur within a given service stub. Multiple stubs can exist within a single system service, but typically only one or two.

StateSpy is very easy to use with any Java-based system and supports Android system services out of the box; it can automatically detect quiescence, transaction entry points, and from which application a given transaction originates. Over 96% of services in Android follow the same design structure: a main service class implements one or more Binder IPC stub interfaces, which are auto-generated by Android's build system from Interface Description Language specifications. A stub is an abstract class that most services either directly inherit from or implement as an anonymous inner class; both variants are detectable by StateSpy. Once a stub class is located, we simply assign the Binder `onTransact()` IPC handler as its entry point and monitor it for incoming transactions. All data was collected from stock AOSP running Android Marshmallow 6.0.1 on a Nexus 5 smartphone.

6.2 State Spill is Prevalent in Android

Our holistic analysis of Android system services reveals the ubiquity and extent of state spill, as depicted in Figure 4. State spill is so deeply embedded in Android services that our tool detected no state spill in less than 6% of the 100+ transaction-handling stubs (AIDL interfaces) we analyzed. We present results on a per-stub basis instead of per-service for granularity and precision reasons, though most services only implement a single stub. Though 76% of service stubs have fewer than 10 *distinct spilled states*, some of the larger stubs have far more, which typically scales with the complexity of the implementing service. A distinct spilled state refers to a single member field in the service entity that was changed by a transaction handler.

Some state spill is relatively straightforward but still hinders various computing goals. For example, callbacks and configuration settings render **process migration** and **virtualization** of those services infeasible. StateSpy detected state spill in Android's `ClipboardService` and `AlarmManagerService` (among others) that jeopardizes **fault tolerance**. When these services restart after a crash, they no longer function properly — clipboard copy/paste and alarms cease to work — and cause user applications to fail mysteriously. If hardened against state spill, these services can be properly restored post-failure to preserve application correctness[17].

Other state spill instances are more unexpected and have complicated implications difficult to observe with manual inspection alone. For example, a secure application can prompt the user to input his/her password by issuing the `verifyUnlock` transaction to the `KeyguardService`. At first glance, one would think that such a simple transaction is free from state spill; however, StateSpy reveals that the `KeyguardService` causes 3 instances of state spill — in the form of boolean status variables — while handling the `verifyUnlock` transaction. This may represent a potential **security** issue: after a password prompt sets these boolean values, a precise timing attack could allow a malicious application to bypass its own prompt if the original prompt crashes, because those spilled values remain.

Another adverse effect of state spill is the hidden breakage of functionality that applications rely on to ensure security. For example, many banking applications interact with and spill states into the `AlarmManagerService` in order to impose a timeout-based automatic logout, which protects the user's identity and private information in the event of a stolen or misplaced device. A **security** problem arises if that service fails: the timeout will never occur and the application will not automatically log out, despite the application being unaware of the failure and still expecting the timeout to trigger.

To dig deeper into state spill in Android services, we randomly selected 60 service transactions from 21 services (the same set from §5.5) for manual classification and analysis. We classified each state spill instance into one or more of the four categories from §4, based on the semantics of the service code that causes the spill. These categories are not mutually exclusive; for example, all callbacks spilled into a dispatcher service can be considered communication-related, but only a subset of those are for multiplexing purposes. First, state spill in indirection layers is the most common (39%) because many Android services exist *solely* to provide a simpler high-level abstraction of a lower-level feature. Second, state spill in multiplexers is the least common (21%) because most I/O device multiplexing is implemented in Android's hardware abstraction layer (beneath the service layer). Third, state spill in dispatchers is quite common (36%) because Android offers many avenues of communication between applications and services, many of which follow a dispatcher-heavy event-driven programming model. Finally, inter-entity collaboration is a fairly common cause of state spill (23%) because many Android services work together to achieve shared goals, necessitating the tight synchronization described below.

6.3 Primary and *Secondary* State Spill in Android

Our analysis thus far has focused on understanding state spill between a pair of entities, from one source to one destination. Interestingly, we discovered that in addition to such *primary* state spill, we also observed *secondary* state spill in Android from the original destination service to another service. This inter-service spill occurs when one service incites change(s) in both its state and in the state of another service while handling a transaction.

Definition 7. Secondary state spill occurs when the handling of one transaction in entity $\mathtt{D}_1$ results in state spill from $\mathtt{S} \rightarrowtriangle \mathtt{D}_1$ and also triggers another transaction that spills state from $\mathtt{D}_1 \rightarrowtriangle \mathtt{D}_2$. The states spilled from $\mathtt{D}_1 \rightarrowtriangle \mathtt{D}_2$ may originate from either $\mathtt{D}$ or $\mathtt{D}_1$.

We found 52 transactions across 27 system services that cause secondary state spill, a lower bound. Figure 5 shows a select subset of these services and depicts only state spill, not dependence or usage relationships. This graph highlights the "hot" services that harbor the most state spill, indicating which ones stand to benefit the most from a software redesign.

As an example of secondary spill, applications that interact with the `UiModeManagerService` to change the UI will spill view mode configuration state into that service. That service then causes secondary state spill by inciting a change in both the visibility state of the `StatusBarManagerService` and the notification content state of the `NotificationManagerService`. Similar events transpire when an application displays a notification via the `NotificationManagerService`, which spills state into the `VibratorService` and `AudioService`. These forms of state spill are serious obstacles to **live updating** or **hot-swapping** these services, not to mention a reduction in **maintainability**.

6.4 Flux Case Study

To further demonstrate the utility of StateSpy, we compare its results against Flux's manual augmentation of Android system services to support application migration. Flux [55] requires *decorator methods* that selectively record and replay the side effects of a service method (transaction handler), effectively addressing the problem of state spill in those methods, as shown below. The authors of Flux shared with us their full list of decorated methods, which we manually analyzed to determine the ground truth of state spill in each method. We developed an instrumentation tool to trace all Binder transactions invoked by a sample set of applications and ran StateSpy on each transaction. To ensure a fair comparison, we conducted this study using the same version of Android 4.4.2 and the same set of applications used in Flux's evaluation, we ignored all graphics-related services because Flux circumvents the need for migrating graphics service states, and we ignored the `ActivityManagerService` because Flux heavily customizes its behavior for migration purposes.

Our evaluation captured 113 unique Binder transactions across 39 unique service stubs. We manually analyzed all 113 transactions to evaluate StateSpy's accuracy and found 1 false positive and 3 false negatives, caused by incomplete handling of native methods. Of the 113 transaction methods, 26 were supported by Flux decorators, while 87 were not. Interestingly, 24 of those 26 decorated methods harbor state spill, highlighting the strong correlation between state spill in service methods and the need to understand and augment those service methods (i.e., handle residual dependencies) to ensure correct post-migration behavior.

In addition, StateSpy detected true state spill in 18 of these unsupported 87 methods, indicating potentially incorrect behavior in a Flux-migrated application. For example, state spill in the `AppWidgetService` can cause application widgets on the homescreen to no longer receive and dis-

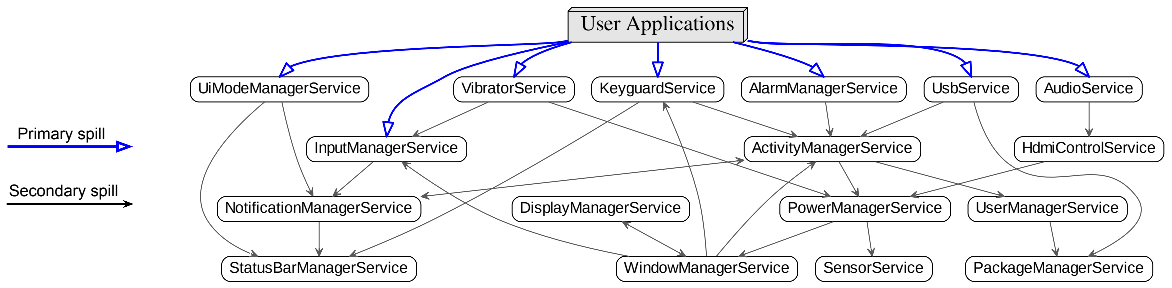

Figure 5: A selected subgraph of complex primary and secondary state spill between applications and system services in Android.

play updates from the migrated application. State spill in two `TextServicesManagerService` methods could cause (*i*) predictive text suggestions to fail when typing, and (*ii*) a loss of Binder death notifications, meaning that applications would not know if the service had failed. The `PackageManagerService` harbors state spill that may result in application components (e.g., services, content providers, activities) that were enabled/disabled on the source device not being properly enabled or disabled on the target device post-migration. Likewise, state spill in the `NfcService`'s `setAppCallback()` method can cause NFC-reliant migrated applications to lose communication with the target device's NFC radio. While it is of course possible for Flux to add support for these 18 methods, this case study demonstrates the utility of the STATESPY tool in identifying transaction methods and spilled states that hinder application migration.

7. Design Recommendations for State Spill

Our results paint a very pessimistic picture: state spill is prevalent and deep in operating systems. *Is it possible to eliminate state spill?* Although a complete answer to this question is beyond the scope of this paper, we suggest solutions for removing or reducing the effects of state spill, based on design pattern classification of §4. We combine the discussion of multiplexers with indirection layers and dispatchers with collaboration (communication) because their solutions are related.

7.1 State Spill-free Indirection and Multiplexing

Client-provided Resources: Many client-server entity designs mandate that servers provide resources (e.g., memory allocation) for clients to request, a pattern across microkernel servers and Android services. For example, Android's `SurfaceFlinger` graphics service offers graphics buffers to each application upon request, constituting state spill.

We argue that the onus of providing memory space for server operations should be on the *client*, forcing the client to own and supply the resource within its entity bounds. Note that this does not preclude the server entity from setting permissions for that resource, e.g., a read-only memory region in the client's address space that the server can modify. This pattern vastly reduces or removes state spill because no state is stored or changed in the server; thus, no matter what one client requests, the server cannot harm or use any other client's resources. As mentioned in §4.2, Genode's Nitpicker WM [21] follows this pattern by requiring clients to allocate memory for server-managed view windows, à la CuriOS [16].

Separating Multiplexing from Indirection: Resource multiplexing functionality typically coexists with indirection providers in a single entity, e.g., most device drivers, conflating their responsibilities. We argue that these two should be separated into physically distinct entities. A dedicated multiplexer allows the underlying driver entity to be free from state spill if implemented correctly (e.g., using the stateless protocols below). State spill in a dedicated multiplexer is difficult to solve, but can be vastly reduced through designs that force client-owned resources (above). Spooling is a good example of extracting multiplexing functionality into an independent entity. Multiple client applications that wish to print documents send them to a spooling directory (multiplexer), after which only a single printer driver (indirection layer) accesses the spooled documents and writes them out to the printer device. While there may be state spill temporarily during the holding phase, those states are soon externalized once the driver prints the documents.

Hardening Entity State: States in a software entity are somewhat ephemeral; their lifetimes are dependent upon the volatility of their underlying storage allocation, e.g., registers, memory, or secondary storage. By decoupling a state's lifetime from the lifetime of its containing entity, that state becomes *hardened* — if the entity is destroyed or recreated, the hardened state(s) will remain. Thus, once a state is hardened, it no longer contributes to state spill. State hardening is achieved most easily through externalization, e.g., a filesystem cache hardens state by writing it to disk; other approaches also exist, such as moving states out of the entity's address space. In fact, we have explored retrofitting Android system services with such a state hardening mechanisms to reduce the effects of state spill and improve fault tolerance [17]. Another example of state hardening is X's session manager, `xsm`, which stores window configuration and organization details persistently to preserve a user's windows across logins.

7.2 State Spill-free Communication

Modularity without Interdependence: Modularity often conflates the separation of concerns with decoupling and independence, but as we have shown, modularity alone does not guarantee a reduction in coupling or interdependence, nor does it preclude state spill. MINIX 3's servers and other entities from §4.3 fall victim to this modularity predicament. In order to reduce state spill, modular entities that have tightly-coupled dependencies should be avoided in favor of loosely-coupled ones that communicate over stateless protocols (below). Although state spill may be viewed as a form of data and control coupling [36], coupling is merely necessary but not sufficient for state spill (§8). If interdependence or coupling is unavoidable, system designers should avoid replicating states that must be propagated throughout the system upon synchronization events; instead, states should be orthogonal. When state replication is required, e.g., caching, one entity should be able to track the spill of its states into other entities such that replicas can be updated.

Clean Communication Models: A straightforward way to reduce state spill is to avoid communication protocols that rely on prior communication. For example, RESTful communication [22] used in the web requires stateless protocols, i.e., *self-sufficient* transactions: all information necessary for handling a request must be supplied with that request, alleviating the web server from storing client-related data that becomes state spill. We realize that stateless protocols may sometimes be impractical, especially at lower abstraction levels, as protocols operate at the coarsest possible granularity. Despite this, there are still valuable takeaways: (*i*) self-sufficient transactions are preferable, and (*ii*) client entities should take responsibility for tracking their own server-side progress.

Additionally, universal broadcast mechanisms can reduce state spill by treating each client equally, obviating the need for a server to maintain client-specific state. For example, Lamport's bakery algorithm [30] allows a server to avoid storing a client queue by assigning each client a number and then calling them one by one for access. This design is clearly beneficial for reducing state spill in dispatchers and is applicable to publisher-subscriber models as well.

8. Related Work

Although we are the first to identify and study the problem of state spill to the best of our knowledge, many prior works have recognized its challenges but failed to identify it specifically. Instead, they often go to great lengths to treat or circumvent the symptoms of state spill. As the above sections deeply covered such works, we now focus on works that address phenomena related to but different than state spill.

Modularity: As state spill occurs between two entities or modules, it is clear that modularity is a *prerequisite*, not a *solution*, for state spill. Therefore, efforts toward improving OS modularity, e.g., Flux OSKit [24], Knit [42], THINK [20], and OpenCom [15], do not by themselves reduce state spill. Our work opens the door to investigating a special form of modularity where no state spill occurs between modules.

Module Coupling: On a related note, software engineering works have studied and classified various forms of coupling and dependency interactions between modules [36, 44, 58, 33, 43]. Although state spill often falls into one of the six types of data and control coupling in [36], coupling is concerned with the transfer of data and control between entities, while state spill measures its lasting effects; thus, coupling is necessary but not sufficient for state spill to occur. For example, a parameter that only affects temporary variables, such as `int b` on L9 of Listing 1, causes coupling but not state spill; thus, it has no lasting effect on the entity and does not impact the aforementioned computing goals.

Information Flow: State spill is related to taint tracking and other information flow control techniques, well-studied topics in systems software [59, 29, 60] and more recently Android [8, 18]. While these works use similar detection techniques as StateSpy, they are primarily concerned with the security/privacy implications of data propagating through a system and leaking from a flawed entity, not whether that data induces an impactful change in the entities to which it propagates. Other works explore the bigger picture of how interactions between multiple entities (Android components) can cause security vulnerabilities [13, 35, 56]. These tools do not identify state spill, but are complementary to StateSpy and could improve its scope by revealing a finer-grained, more detailed chain of interactions between source, intermediary, and destination entities.

Designs that may reduce state spill: Without explicitly recognizing the problem of state spill, many have sought to compartmentalize important states in applications, e.g., Microreboot [12], and in OS components, e.g., CuriOS [16] and Barrelfish/DC [61]. These designs may unknowingly reduce state spill but are not easily applicable to existing large-scale OSes like Android. Library OSes allow user applications to specify or provide their own implementation of a given service or OS feature for individual use [19, 41], potentially reducing state spill. However, a private service per application is inefficient, and once that custom service implementation is utilized by other applications, it becomes susceptible to the same state spill as other OS designs. Similarly, the Unikernel design philosophy packages applications, services, and the kernel into a single sealed appliance [31], bypassing state spill entirely because everything becomes a single coarse-grained entity. However, this design renders the aforementioned computing goals either very difficult or entirely impossible due to a lack of software flexibility — those goals all require loosely coupled, independent, and replaceable OS entities. As previously mentioned, RESTful architectures used in web services [40, 39] address state spill at the highest possible granularity level, but are irrelevant to computing goals within a single machine. Olsson [37] applied RESTful principles to a very simple client-side API with mixed results, focusing more on client-side design aspects rather than its effects on the server's state and behavior.

9. Concluding Remarks

In this work, we define and identify state spill as the underlying reason why important computing goals such as process migration, fault isolation and recovery, live update, and hot-swapping have been difficulty to realize. We show that important design patterns in system software are susceptible to state spill and develop the StateSpy tool to help developers automatically identify instances of state spill in large-scale software systems. By applying StateSpy to Android system services, we show the prevalence and deep nature of state spill therein. We believe that as software systems grow in complexity and size, state spill lurks underneath as a bottleneck to reliability, security, and scalability. With this work, we hope to bring state spill to the community's attention as an understudied, important phenomenon in software systems.

ACKNOWLEDGMENTS

We thank the authors of Flux [55] for sharing unpublished results, our shepherd Dr. Dilma Da Silva, and the anonymous reviewers for their feedback. This work was supported in part by NSF Award CNS #1422312 and its REU supplement.

References

[1] Design of the Genode OS Architecture: Interfaces and Mechanisms. http://genode.org/documentation/architecture/interfaces. Accessed: 2016-02-22.

[2] Invoking user-space applications from the kernel. https://www.ibm.com/developerworks/library/l-user-space-apps/. Accessed: 2016-10-13.

[3] UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html. Accessed: 2016-10-10.

[4] X.Org Security Advisory: Dec. 9, 2014. https://www.x.org/wiki/Development/Security/Advisory-2014-12-09/. Accessed: 2016-10-11.

[5] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. Opus: Online patches and updates for security. In *Proc. USENIX Security*, 2005.

[6] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: A virtual mobile smartphone architecture. In *Proc. ACM SOSP*, 2011.

[7] J. Arnold and M. F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proc. ACM EuroSys*, 2009.

[8] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. ACM PLDI*, 2014.

[9] P. Barton-Davis. Linux kernel mailing list: upcalls. http://lkml.iu.edu/hypermail/linux/kernel/9809.3/0922.html. Accessed: 2016-10-13.

[10] K. Boos. StateSpy: State Spill Characterization Project Website. http://download.recg.org.

[11] K. Boos, A. Amiri Sani, and L. Zhong. Eliminating state entanglement with checkpoint-based virtualization of mobile OS services. In *Proc. ACM APSys*, 2015.

[12] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *Proc. USENIX OSDI*, 2004.

[13] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proc. ACM MobiSys*, 2011.

[14] D. D. Clark. The structuring of systems using upcalls. In *Proc. ACM SOSP*, 1985.

[15] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems*, 2008.

[16] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving reliability through operating system structure. In *Proc. USENIX OSDI*, 2008.

[17] P. Dong. Reducing fate-sharing in software systems via fine-grained checkpoint and restore. Master's thesis, Rice University, 2016.

[18] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. USENIX OSDI*, 2010.

[19] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. ACM SOSP*, 1995.

[20] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller. THINK: A software framework for component-based operating system kernels. In *Proc. USENIX ATC*, 2002.

[21] N. Feske and C. Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. Technical report, Technische Universität Dresden, 2005.

[22] R. Fielding. Representational state transfer. *Architectural Styles and the Design of Network-based Software Architecture*, 2000.

[23] R. T. Fielding and R. N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2002.

[24] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proc. ACM SOSP*, 1997.

[25] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Safe and automatic live update for operating systems. In *Proc. ACM ASPLOS*, 2013.

[26] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 2006.

[27] K. Hui, J. Appavoo, R. Wisniewski, M. Auslander, D. Edelsohn, B. Gamsa, O. Krieger, B. Rosenburg, and M. Stumm. Supporting hot-swappable components for system software. In *Proc. ACM HotOS*, 2001.

[28] A. Kadav, M. J. Renzelmann, and M. M. Swift. Fine-grained fault tolerance using device checkpoints. In *Proc. ACM ASPLOS*, 2013.

[29] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. ACM SOSP*, 2007.

[30] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 1974.

[31] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proc. ACM ASPLOS*, 2013.

[32] K. Makris and K. D. Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proc. ACM EuroSys*, 2007.

[33] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proc. ACM/IEEE ICSE*, 2000.

[34] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Comput. Surv.*, 2000.

[35] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication

mapping in android with EPICC: An essential step towards holistic security analysis. In *Proc. USENIX Security*, 2013.

[36] A. J. Offutt, M. J. Harrold, and P. Kolte. A software metric system for module coupling. *Journal of Systems and Software*, 1993.

[37] R. Olsson. Applying REST principles on local client-side APIs. Master Thesis, KTH Royal Institute of Technology, 2014.

[38] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proc. USENIX OSDI*, 2002.

[39] C. Pautasso and E. Wilde. Why is the web loosely coupled?: a multi-faceted metric for service design. In *Proc. WWW*, 2009.

[40] C. Pautasso, O. Zimmermann, and F. Leymann. RESTful web services vs. 'big' web services: making the right architectural decision. In *Proc. WWW*, 2008.

[41] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library os from the top down. In *Proc. ACM ASPLOS*, 2011.

[42] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proc. USENIX OSDI*, 2000.

[43] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proc. ACM OOPLSA*, 2005.

[44] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and A. J. Offutt. Maintainability of the linux kernel. *IEE Proceedings-Software*, 2002.

[45] R. W. Scheifler and J. Gettys. The x window system. *ACM Transactions on Graphics (TOG)*, 1986.

[46] K. Sen. Concolic testing. In *Proc. IEEE/ACM ASE*, 2007.

[47] R. Sethi. *Programming languages: concepts and constructs*. 1996.

[48] M. Siniavine and A. Goel. Seamless kernel updates. In *Proc. IEEE/IFIP DSN*, 2013.

[49] C. A. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. Da Silva, G. R. Ganger, O. Krieger, M. Stumm, M. A. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proc. USENIX ATC*, 2003.

[50] B. P. Swift. User mode scheduling in MINIX 3. Technical report, Vrije University Amsterdam, 2010.

[51] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proc. USENIX OSDI*, 2004.

[52] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proc. ACM SOSP*, 2003.

[53] P. Tullmann, J. Lepreau, B. Ford, and M. Hibler. User-level checkpointing through exportable kernel state. In *Proc. IEEE Wrkshp. Object-Orientation in Operating Systems*, 1996.

[54] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proc. CASCON*, 1999.

[55] A. Van't Hof, H. Jamjoom, J. Nieh, and D. Williams. Flux: Multi-surface computing in Android. In *Proc. ACM EuroSys*, 2015.

[56] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proc. ACM CCS*, 2014.

[57] M. Weiser. Program slicing. In *Proc. ACM/IEEE ICSE*, 1981.

[58] L. Yu, S. R. Schach, K. Chen, G. Z. Heller, and J. Offutt. Maintainability of the kernels of open-source operating systems: A comparison of Linux with FreeBSD, NetBSD, and OpenBSD. *Journal of Systems and Software*, 2006.

[59] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. USENIX OSDI*, 2006.

[60] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing distributed systems with information flow control. In *Proc. USENIX NSDI*, 2008.

[61] G. Zellweger, S. Gerber, K. Kourtis, and T. Roscoe. Decoupling cores, kernels, and operating systems. In *Proc. USENIX OSDI*, 2014.