Eliminating State Entanglement with Checkpoint-based Virtualization of Mobile OS Services

Kevin Boos Ardalan Amiri Sani Lin Zhong

Rice University, Houston, Texas

{kevinaboos, ardalan, lzhong}@rice.edu

Abstract

Mobile operating systems have adopted a service model in which applications access system functionality by interacting with various OS Services in separate processes. These interactions cause application-specific states to be spread across many service processes, a problem we identify as *state entanglement*. State entanglement presents significant challenges to a wide variety of computing goals: fault isolation, fault tolerance, application migration, live update, and application speculation. We propose CORSA, a novel virtualization solution that uses a lightweight checkpoint/restore mechanism to virtualize OS Services on a per-application basis. This cleanly encapsulates a single application's service-side states into a private virtual service instance, eliminating state entanglement and enabling the above goals. We present empirical evidence that our ongoing implementation of CORSA on Android is feasible with low overhead, even in the worst case of high frequency service interactions.

1. Introduction

Modern mobile operating systems, such as Android, iOS, and Windows 8 RT, provide a plethora of *OS Services* to serve applications. These services often run as threads or managed runtime objects in special userspace processes, using IPC to provide applications with system functionality, e.g., I/O access, UI management, and media processing in Android.

Compared to the traditional shared library model, in which an application dynamically loads library functions into its address space, the OS Service model offers many benefits. For example, it offers a more modular design that improves maintainability, bolsters security enforcement by reducing the attack surface, and simplifies the incorporation of closed-source, third-party services. In fact, the system resulting from this model, e.g., Android, closely resembles a microkernel-based operating system [16] and enjoys many of the same design benefits.

However, the proliferation of OS Services has resulted in the violation of a long-standing convention: *all states* of an application are encapsulated in its own process memory. Although the service model encourages separation of concerns, its implementation bears a hidden cost: application-relevant states are now strewn throughout the OS, spread across multiple service processes. For example, many services in Android store a list of application callbacks that have been registered to trigger upon some event, such as a new sensor or location input. More complex services like the camera service and SurfaceFlinger, Android's graphics manager, allocate and maintain buffers on behalf of applications to store data like frames and textures.

These service-side states create the *state entanglement* problem: a tight coupling between an application and the underlying OS instance. State entanglement presents significant challenges to the important computing goals detailed in Section 2.2: fault isolation, fault tolerance, application migration, live update, and application speculation.

To eliminate state entanglement, we propose *OS Service virtualization*, in which an application sees its own virtual instance of an OS Service. This approach necessarily decouples applications' service-side states from the OS Service(s) in which they reside, encapsulating a single application's service-side states into its own virtual service instance. Decoupling and encapsulating these states removes the barriers to the above important use cases.

We present our early efforts towards CORSA, the first work to use checkpoint/restore as a means of virtualization, to the best of our knowledge. CORSA virtualizes an OS Service by checkpointing its state such that each checkpoint only contains service-side states for a single application, providing the illusion of per-application service instances. In order to know when to checkpoint/restore, CORSA intercepts all communications between services and applications, ensuring that only a single application interacts with a given service at a time. To better illustrate these semantics, envision a scenario

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

APSys 2015, July 27–28, 2015, Tokyo, Japan..
Copyright is held by the owner/author(s).
ACM 978-1-4503-3554-6/15/07.
http://dx.doi.org/10.1145/2797022.2797041

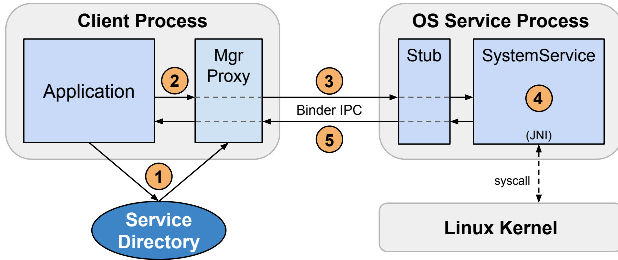

Figure 1. The OS Service interaction model in Android.

in which application `Foo` wishes to access an OS Service already in use by application `Bar`. CORSA saves the current service state as a checkpoint associated with `Bar` before allowing `Foo` to proceed. When `Bar` attempts to access the service again, CORSA saves the current service state as `Foo`'s latest checkpoint and restores `Bar`'s latest checkpoint to the service. Thus, from the application perspective, both `Bar` and `Foo` see their own private instance of the service, while only one service instance exists from the system perspective. Section 3 describes this process in further detail.

We discuss our implementation plans in Section 4 and present a feasibility study of CORSA in Section 4.3. We demonstrate that a checkpoint/restore mechanism is efficient enough to be feasible for OS Service virtualization and that it introduces no user-perceivable overhead, even in the extreme cases of rapid and periodic transactions.

While this paper targets OS Services, our checkpoint-based virtualization approach is also applicable to a wide variety of domains, e.g., implementing generic OS-level virtualization through kernel state checkpointing, or isolating and virtualizing microkernel servers. A notable advantage of our method over others is that checkpointing does not require any knowledge or understanding of the entity being virtualized; for example, OS-level virtualization using checkpointing would not require knowledge of the kernel's system calls. CORSA can even virtualize closed-source services like Google Services and vendor-specific daemons.

2. Background: OS Service Model

Mobile OS Services do not execute in the same process as the applications that use them, so accessing their functionality is more involved than with a shared library model. Figure 1 shows the procedure that an application undergoes to access an OS Service in Android:

1. The application queries the service directory, indexed by service name. The service directory returns a *Manager* proxy interface to the OS Service.
2. The application calls a Manager convenience function with the arguments it wishes to pass to the OS Service.
3. The Manager transparently marshals the function and its arguments into a Parcel and initiates a Binder IPC *transaction* to send the serialized Parcel to the OS Service.
4. After the OS Service's Stub unmarshals the function arguments from the Parcel, it verifies application permissions and then executes the remote function, which may invoke kernel syscalls or other services.
5. The Stub returns the function's result to the Manager proxy, which delivers it back to the application as if it were a local function call all along.

Each application has its own private instance of a Manager proxy for every OS Service it uses. This 1:N mapping between an OS Service and its Manager proxies forces the service to act as a multiplexer, implicitly storing application-specific state about its connections with each proxy.

2.1 Classification of Service Architectures

Unlike desktop OS daemons, each mobile OS Service is not cleanly contained within its own process, or even within its own thread. Android implements OS Services in three ways across different layers of its userspace software stack.

`SystemServices` are implemented as Java classes that run in a managed environment: the Dalvik VM (now ART). All `SystemServices` are instantiated upon boot as singleton objects in the `system_server` process, sharing one address space. They can spawn their own thread — though most do not — but not their own process. These services comprise the frameworks behind Android's main SDK, frequently interacting with applications and other services.

Native Core Services are implemented as C/C++ libraries that handle requests from upper-level services or applications through Binder IPC. These services typically spawn their own thread and live in one of several container processes, i.e., *mediaserver*, *surfaceflinger*, or *sensorservice*, but never within the `system_server` process.

Native Daemons each execute in their own process and are started by `init` upon boot, but use standard Unix domain sockets in place of Binder IPC. They are not directly available to user applications in Android.

2.2 Problems with OS Service Model

The distributed nature of the OS Service model causes *state entanglement*, i.e., application-specific states being stored in the service's process memory. These states may be explicit, e.g., data that the application has previously passed to the service for future use, as well as implicit, e.g., a representation of the current connection between an application and service. Both the application and service rely on the existence of these service-side states for correct operation going forward, posing a challenge to a variety of use cases.

For **fault isolation**, if an OS service crashes when serving one application, other applications or services using that service will experience failures as well. This introduces new security threats: if a malicious application compromises a service, e.g., by exploiting bugs, it can potentially affect other applications relying on that service by either accessing their data or causing a denial of service.

For **fault tolerance**, recovering from a failure on behalf of one application requires restoring the service to a prior

state, which unexpectedly breaks all other applications using that service. Proper fault isolation is a prerequisite for side effect-free fault recovery.

For **application migration**, the application process is transferred to and resumes execution on a new device. The application, unaware that it has migrated, will attempt to continue using an OS Service from the old device, but will immediately fail because the service on the new device lacks the old states needed to handle that application's requests. Application migration requires transferring not only the application process itself but also its states in other service processes [24]. Identifying and extracting these service-side states requires expert knowledge and explicit per-service support, woefully absent from today's services.

For **live update** of an OS Service, the states in that service processes must reach quiescence [10] before being saved; live updating a long-running application requires quiescence in both its process state and that of its services, not easily achieved compared to traditional OSes.

Finally, for **application speculation**, the OS must properly manage each speculative instance's service-side states, adding them before forking it and removing them after forking it based on speculation correctness. Pruning a specific instance's states from a given service is difficult without affecting other applications using that service.

Properly encapsulating application-specific state in OS Services is the key to eliminating state entanglement. In the following section, we explain our approach for doing so.

3. Design of CORSA

In order to solve the state entanglement problem identified above, we posit that OS services should be virtualized on a per-application basis. Each virtual service instance serves one application only, cleanly encapsulating that application's service-side states.

We present CORSA, our solution for virtualizing OS Services that enables the following use cases. For fault isolation, an OS service failure only affects that specific virtual instance, isolating it from other applications. This also facilitates fault tolerance, making it possible to restore a previous version of the service without affecting other applications. Application migration simply requires transferring its virtual service instance(s) to the target machine. Transferring an OS Service *without* virtualization is destructive because it would overwrite that service's states for all other applications, causing everything but the newly-migrated application to fail. Live update of both applications and services are simplified because quiescence is only required for a single application-service pair at a time. Finally, application speculation is much easier because adding/removing service-side states for newly forked instances becomes trivial.

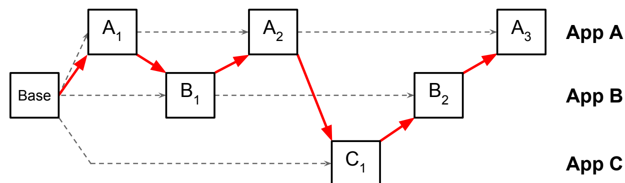

Figure 2. The *checkpoint tree* for one OS Service. Each node represents a checkpointed service instance for a given application; X_N is the Nth checkpoint for application X. The dashed arrows convey a history of checkpoints; solid arrows represent a transition between applications using the service.

3.1 Checkpoint-based Virtualization

CORSA employs an innovative checkpoint-restore mechanism that encapsulates the service-side states of each application into private virtual service instances. Our key idea is to time-multiplex the service between applications such that it serves only one application at a time, switching between virtual service instances as needed. At switch time, CORSA checkpoints the current application's service instance and restores the next application's most recent service checkpoint. CORSA monitors and intercepts IPC between applications and services to know when to checkpoint/restore. This procedure is conceptually similar to a context switch between multiple threads running on a CPU core.

In this way, we guarantee that CORSA preserves two key invariants: (*i*) only a single application accesses a given OS Service at a time, and (*ii*) each virtual instance checkpoint contains service-side states that belong to *one application only*, eliminating state entanglement.

Feasibility of Checkpointing OS Services

An immediate concern is the efficiency of checkpoint/restore operations and whether they will affect the user experience. A checkpoint/restore must occur every time applications alternate using a given OS Service, so the frequency of application switching determines the time frame for checkpointing. In order for checkpointing to be user-imperceptible, the minimum time between two different applications transacting with a given service must be greater than the time required to checkpoint/restore that service. Our measurements in Section 4.3 indicate that checkpoint-based virtualization is not only feasible but has minimal impact on transaction latency and application responsiveness, even in the atypical worst case of very frequent application switching.

Furthermore, in Section 4.1, we show that checkpoint and restore operations need not execute sequentially and can be done in parallel, further reducing the critical path between application transitions to that of just a restore operation.

Checkpoint Management

As shown in Figure 2, CORSA maintains a *checkpoint tree*, a collection of checkpointed virtual service instances organized

by client application. Figure 2 also illustrates the temporal progress of an OS Service. It depicts a scenario where the service handles transaction requests from three applications in the following order: A, B, A, C, B, A. For the first transaction from a given application, CORSA restores the service to its Base checkpoint, providing the application with a fresh virtual service instance. When the service receives a transaction from a different application, CORSA checkpoints the service, adds it to the checkpoint tree, and then restores the latest checkpoint associated with the next application. Note that we describe checkpoint and restore as ordered, sequential operations for conceptual clarity; they do not necessarily have to occur sequentially or in this order (see Section 4.1).

To minimize memory consumption, CORSA reduces checkpoint trees using delta compression and prunes old nodes from the tree according to an application-adjustable policy. There is an inherent trade-off between memory usage and fault tolerance: if fault tolerance is of higher priority, CORSA preserves a longer checkpoint history tailored to the application's needs; otherwise, CORSA retains only the most recent service checkpoint for each application.

3.2 Design Challenges

The design of CORSA thus far faces two notable challenges. First, certain OS Services require simultaneous access to states from more than one application, in direct conflict with the invariants guaranteed by per-application service virtualization. Second, many services interact with external entities, such as I/O devices or cloud servers. Applying a naïve checkpoint/restore mechanism will break the services' interactions with such entities. We describe our plan to solve these two challenges below.

Services Requiring Multiple Apps' States

Some OS Services need access to multiple applications' states simultaneously in order function correctly. For example, Android's `ActivityManagerService` (`AMS`) is responsible for starting and delivering lifecycle events to applications, navigating between Activities (UI windows), generating the "recently used apps" list, and more. In this case, applying CORSA's normal checkpoint-based virtualization mechanism to the *entire* service would cause the service to behave incorrectly; for example, the `AMS` would only ever show a single application on its recent apps list.

Our key insight into supporting these services is that they need not be virtualized in their entirety. We can refactor them into a per-application *service frontend* and a single, system-wide *service backend*. Each frontend contains an individual application's state plus any functionality that needs access to *only* that state, such as `AMS`'s lifecycle events and Activity navigation. The backend only contains functionality that requires access to multiple applications' states at once, e.g., the `AMS`'s recent apps list. To ensure that frontends are fully decoupled from the backend, the backend must be stateless. We maintain a stateless backend by querying every available frontend in a uniform, agnostic fashion, only when the backend needs to gather multi-application states. We then apply CORSA's checkpointing approach to virtualize each frontend only, thus preserving state isolation and maintaining our two invariants.

Services Interacting with External Entities

Many OS Services are not simply self-sufficient; they often communicate with other external entities, e.g., I/O devices and cloud servers. One determining factor in an OS Service working with CORSA is whether its external entities are able to *multiplex*, i.e., accommodate multiple service instances at once. If an entity cannot multiplex, then CORSA may violate the entity's expectations by creating multiple service instances and switching between them without the entity's knowledge. For example, CORSA will work with I/O-related OS Services as long as their external entities, device drivers, allow multiple service instances to exist. In Section 4.2, we show that existing device drivers either support multiplexing or can be easily modified to do so.

Another important consideration is whether such external entities are *stateful* or *stateless*. An entity is stateful if it maintains some state for each service instance communicating with it, and stateless if it does not. Stateless entities present no difficulties, but stateful entities require additional consideration for the following cases.

For fault isolation and tolerance, a crashed OS Service instance must not corrupt the external entity, and the entity should clean up any residual state belonging to the crashed service instance. The same requirements hold true for live update and application speculation, where service-side states must not violate consistency with the external entity, either post-update or post-speculation. In the case of Android's sensor service, this procedure is as simple as closing the file descriptor of the sensor device file, which causes the kernel to invoke the driver's cleanup handlers. For the camera service, an explicit API call into the driver is needed to reset the camera. Section 4.2 provides further details.

For application migration, an OS Service instance must retain the proper state to continue interacting with its entity post-migration. If the entity is a cloud server, we can achieve this by migrating the network state of the service-cloud connection, à la [13]. However, if the entity is local, like an I/O device, its state needs to be either migrated to or recreated on the target device. Since I/O migration requires explicit hardware/driver support, typically non-existent, we opt to recreate the state of the I/O device using a record and replay technique inspired by [9, 18]. That is, we record the interactions between the OS Service and device driver on the host machine and then replay them on the target machine.

Recording and replaying service interactions incurs storage overhead on the order of the number of recorded events. We reduce this overhead by clearing the recorded logs whenever the I/O device reaches a stateless point. In the camera

service example, we can reset the log whenever the application releases its camera handle, which turns off the camera.

4. Ongoing Implementation

Although we regard the choice of checkpointing software as an implementation decision orthogonal to the design of CORSA, existing works suffer from the limitations explained in Section 6. Thus, we introduce our preliminary, ongoing implementation of checkpoint-based virtualization on Android.

4.1 Checkpoint/Restore Mechanism

We adopt a lightweight approach that checkpoints an OS Service by duplicating its address space, registers, and other process structures from within the kernel. This takes full advantage of Linux's copy-on-write feature to significantly reduce initial checkpointing delay. Our restore implementation is even faster, simply swapping pointers for the current process control block to those of a prior checkpoint.

We identify a latency-reducing optimization opportunity: *only the restore operation must occur during an application switch*, the checkpoint operation can actually execute in parallel or asynchronously after a prior checkpoint is restored, due to the disjoint nature of the two memory regions. As such, the requirements for checkpoint and restore are different; checkpointing can be slow but restoring should be as fast as possible. This optimization shortens the critical path of an application switch from checkpoint + restore to just restore.

CORSA intercepts IPC between applications and an OS Service by instrumenting Android's Binder kernel driver. CORSA monitors both the application proxy side and the service stub side of the driver (§2) in order to track which application issued the transaction and which service should be checkpointed. Checkpoint/restore operations are triggered right before the service thread executes the transaction.

`SystemServices` (§2.1) further complicate our implementation because they do not run within their own thread or process, rendering existing JVM checkpointing approaches inapplicable [15]. We plan to inject code into the Java runtime (Dalvik/ART) to duplicate the state of a `SystemService` object (e.g., heap, class files) to an in-memory copy, avoiding the overhead of conventional object serialization [21].

4.2 Handling External Entities

In this section, we investigate whether external entities support multiplexing. We observe three different entity behaviors, consistent across two platforms: a Galaxy Nexus running Android 4.2, and a Nexus 5 running Android 5.0.

(i) Some entities fully support multiplexing. For example, we run two instances of Android's input service, which communicates with the kernel's touch screen device driver. The touch screen driver successfully delivers touch input events to both service instances, reporting no errors.

(ii) Some entities support multiplexing but require one OS Service instance to perform certain actions before switching to another. For example, the camera device driver requires the service using it to close its camera connection before another connection can be opened. To support such entities in CORSA, the service instance merely needs to query the entity and initiate a new connection only when the entity permits.

(iii) Some entities can fully support multiplexing after trivial changes. For example, the sensor device driver has a simple sanity check that prevents multiple services from using it concurrently, merely because it does not expect that behavior by default. Once we disable that check, the sensor driver supports multiple service instances concurrently.

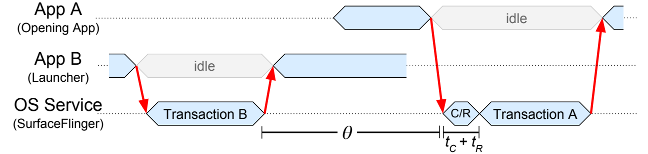

Figure 3. Timing diagram of checkpoint/restore operations when two applications transact with an OS Service.

4.3 Feasibility Analysis

In Section 3.1, we claim that checkpointing an OS Service is a feasible means of virtualization. To justify that claim, we obtain and analyze the following measurements on a Nexus 5 smartphone running Android 5.0: the time to checkpoint (t_C) and restore (t_R) a service, the interval θ between two one-shot transactions, and the maximum frequency f of periodic event transactions.

First, we measure the execution time of a superfluous checkpointing algorithm to be $t_C = 0.3\ ms$ at most, which copies every member of `task_struct` (the kernel's process structure) to an in-memory replica. Note that the actual contents of virtual memory and files are not duplicated, just the structures that represent them. This memory copy is only necessary for each service's initial Base checkpoint; subsequent checkpoints will quickly copy only the deltas.

We measure the time to restore a prior checkpoint to be $t_R = 4.4\ \mu s$ at most. This represents the time to swap the current process's `task_struct` with a previously checkpointed one and activate its new address space. Although a restore operation incurs negligible latency, the new address space introduces minor overhead to the service's next transaction by page faulting on writes to memory, a hidden side effect of copy-on-write. By profiling `fork()`, we find that it takes only $1\ ms$ to copy and write to about 10 MB of memory, or 2500 pages. Although every service is different, we believe that most transactions will write to far fewer than 2500 pages, incurring well under $1\ ms$ of latency, because transactions are small, short-lived functions.

Second, we measure θ, the interval between two transactions from different applications. The *minimum θ value* is $1.07\ ms$, which occurs when opening a new application from the launcher, because the launcher application and the new application rapidly transact with SurfaceFlinger, Android's

graphics service. Out of 600 data points, we find that nearly 90% of θ intervals are larger than 3 ms, dominating the maximum checkpointing latency t_C by an order of magnitude. Figure 3 depicts the timing relationships between these measurements. As long as θ is larger than the time to checkpoint t_C, which holds true for 100% of our measurements, CORSA will cause no slowdown to applications or services.

Third, we measure f, the maximum frequency of periodic event transactions as 221 Hz, which occurs when the sensor service delivers sensor events to applications. This means that sensor data arrives approximately every $\frac{1}{f} = 4.5\ ms$ at the fastest. Note that this high sampling rate is rarely used, 10-20 Hz is typical. We also measure the time required to deliver a sensor event as 27 μs. Since CORSA must checkpoint and restore the sensor service before delivering the event to each application, it can support $\frac{4.5\ ms}{t_C + t_R + 27\ \mu s} \approx 13$ applications concurrently without slowing down sensor event delivery. Even if readings arrived late due to checkpointing latency, this would not break correctness because Android makes no guarantee about the real-time delivery of sensor data.

Based on these encouraging preliminary figures, we believe that checkpointing is a feasible approach for virtualizing OS Services. Even in the most latency-critical situations, the checkpointing delay t_C is completely masked several times over by the θ interval, and rapid periodic transactions like sensor events suffer no delay when serving over a dozen applications. Although we only provide microbenchmarks at this time, we believe they are a good predictor of final implementation performance because the above measured latency values dominate that of other necessary actions, such as bookkeeping of checkpoints. Furthermore, as described in Section 4.1, our full implementation of CORSA will parallelize checkpoint and restore, further reducing the critical path (Figure 3's C/R block) from $t_C + t_R$ down to just t_R.

5. Discussion

Although we limit the scope of this work to the boundary between applications and mobile OS Services, state entanglement manifests between applications and many other software layers: shared libraries, hardware abstraction layers, device drivers and other kernel subsystems, and even external cloud servers. We believe there is no silver bullet to solving state entanglement across all layers; rather, different layers will necessitate different solutions. For example, the application-relevant states stored in device drivers, e.g., low-level configuration data, are fundamentally different than those in cloud servers, e.g., high-level application status; they also operate across different boundaries: system calls vs. network messages.

Another point of consideration is what level of modification we permit to the existing software stack. Currently, we permit CORSA to make little to no modifications to OS Services in order to maintain compatibility with Android's large legacy codebase. If we permit modification to the entities themselves, we will approach techniques similar to those discussed in the above Manual Disentanglement paragraph (§6). If we permit modification to lower layers like the kernel, higher-level entities can be made stateless without being rewritten. If we permit clean-slate design, an OS built from scratch to better manage states could dramatically simplify or entirely eliminate state entanglement between many (or all) software layers. We could even make entities completely stateless, à la RESTful services, as described at the end of Section 6. In the future, we intend to explore alternative boundaries and techniques for disentangling states across various entities, beyond checkpoint/restore or record and replay.

6. Related Work

We organize our discussion of related work by the technique or boundary used to separate stateful entities from each other.

Checkpoint/Restore: Checkpoint/restore is our chosen technique for saving and recreating states in an entity — an OS Service in this case — to realize its virtualization. We do not purport to reinvent a novel checkpoint/restore design, but nevertheless choose to implement our own solution due to existing works' lack of features and compatibility. Our implementation will provide several Android-required features missing from prior approaches, including support for (*i*) managed runtime environments [1, 3, 8], (*ii*) individual threads and multithreaded processes, and (*iii*) non-GNU/Linux OSes or those without full POSIX compliance.

Record and Replay: One close alternative to checkpoint/restore for recreating states is record and replay [9], in which interactions between entities are recorded and then replayed later to recreate interaction states in fresh entity copies. Checkpointing is generally much more efficient than record and replay, both temporally and spatially, because many recorded interactions can be collapsed into a single checkpoint, and many replayed actions can be achieved with a single restore. Flux [24] is one such work that migrates applications by extending CRIU [1] to Android and recording/replaying interactions between applications and services. Each service function must be manually annotated to define its behavior when recording and replaying interactions. In addition, service developers must also write custom proxy functions that adapt replayed interactions to the new target device, necessary for avoiding incorrect post-migration behavior. CORSA's checkpointing approach does not require significant manual effort nor explicit knowledge of each service, addressing the larger goal of state disentanglement, especially in non-migration contexts. Furthermore, while the high cost of Flux's record/replay approach may be masked by the huge delays of application migration, it is prohibitive for local low-latency operations, such as fault recovery, live update, and application speculation, which CORSA can efficiently handle through checkpoint/restore.

Virtualization: Unlike conventional virtualization, CORSA can virtualize a given entity, such as an OS Service, without also virtualizing everything above it. Traditional VMs offer a narrow, clearly-defined boundary between multiple whole operating systems, making them easier to decouple and migrate [6, 14]. However, because VMs operate at such a low boundary, they are too coarse-grained; for example, migrating or restarting a single entity requires migrating or restarting the entire VM, introducing unnecessary overhead despite efforts to reduce the amount of transferred state [22].

Many other works choose higher virtualization boundaries than VMs. Cells [2] virtualizes at the user-kernel boundary to run multiple Android userspace instances atop a single shared Linux kernel. Similarly, Face-Change [11] uses virtualization to present a different kernel "view" (memory contents) to each application. Zap [19] virtualizes at the process boundary, grouping processes into *pods*, virtual containers that can be migrated. Zap's pods are infectious, i.e., if a process in the pod communicates with some other entity, that entity must be included into the pod. These works suffer from the same problems as VMs: everything above the virtualization boundary remains entangled. In fact, a recent work [24] from the authors of Cells and Zap exemplifies that such namespace virtualization methods cannot solve state entanglement.

Manual Disentanglement: Several previous works *manually* modify applications, e.g., Microreboot [5], or OS components, e.g., CuriOS [7] and Barrelfish/DC [27], to compartmentalize important states. The major disadvantage of these works is that manual modification requires significant engineering effort and a deep understanding of each component, impractical for existing large-scale OSes like Android.

Memory Protection: Another category of works isolates entities via special memory regions/mechanisms, e.g., hardware memory tagging [26] or address space customizations [12, 17, 25]. Nooks [23] combines software memory protection with function interception to isolate a driver from the rest of the kernel. Boxify [4] isolates Android applications in virtual sandboxes but does not fully isolate applications from failed services. These works provide the foundations of isolation but cannot identify which entangled states should be isolated, relying on manual identification (and extraction) of states. CORSA avoids this problem altogether by guaranteeing that each virtual service instance contains only one application's state.

Stateless services: One alternative to disentangling states is to avoid making entities stateful in the first place. RESTful services used across the Web take this approach by sending all client state necessary for invoking a remote function to the server [20], every time for every request. In this way, RESTful services store no client state persistently on the server and are easier to migrate, update, or restart. However, the Web layer can afford RESTful protocols because transactions are relatively slow and occur between different physical machines, but IPC and driver layers do not have that luxury.

7. Conclusion

The OS Service model used in modern mobile operating systems scatters application states across many service processes, violating the convention that an application's states are all stored within its address space. We identify this as one incarnation of the *state entanglement* problem, which presents challenges to several important computing goals: fault isolation, fault tolerance, application migration, live update, and application speculation. To address this problem, we propose CORSA, a novel checkpoint-based virtualization solution that encapsulates a single application's service-side state into a private virtual service instance. We also provide empirical evidence that CORSA's checkpoint/restore mechanism can be implemented with low overhead.

Acknowledgments

The work was supported in part by NSF Award #1422312. The authors also thank anonymous reviewers and their shepherd Dr. Peter Druschel for their useful feedback.

References

[1] Checkpoint/Restore In Userspace. http://www.criu.org/Main_Page. Accessed: 2015-04-22.

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

[3] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proc. IEEE IPDPS*, 2009.

[4] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock Android. In *Proc. USENIX Security*, 2015.

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

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. USENIX NSDI*, 2005.

[7] 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.

[8] J. Duell. The design and implementation of Berkeley Lab's Linux checkpoint/restart. Technical report, Lawrence Berkeley National Laboratory, 2005.

[9] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Oper. Syst. Rev.*, 2002.

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

[11] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu. FACE-CHANGE: Application-driven dynamic kernel view switching in a virtual machine. In *Proc. IEEE DSN*, 2014.

[12] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 2007.

[13] A. Kadav and M. M. Swift. Live migration of direct-access devices. *ACM SIGOPS Oper. Syst. Rev.*, 2009.

[14] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proc. ACM HotMobile*, 2002.

[15] J. Lawall and G. Muller. Efficient incremental checkpointing of Java programs. In *Proc. IEEE DSN*, 2000.

[16] J. Liedtke. On micro-kernel construction. In *Proc. ACM SOSP*, 1995.

[17] A. Lindstrom, J. Rosenberg, and A. Dearle. The grand unified theory of address spaces. In *Proc. ACM HotOS*, 1995.

[18] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu. Live migration of virtual machine based on full system trace and replay. In *Proc. ACM HPDC*, 2009.

[19] 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.

[20] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media, Inc., 2008.

[21] T. Suezawa. Persistent execution state of a Java virtual machine. In *Proc. ACM Java Grande Conf.*, 2000.

[22] A. Surie, H. A. Lagar-Cavilla, E. de Lara, and M. Satyanarayanan. Low-bandwidth VM migration via opportunistic replay. In *Proc. ACM HotMobile*, 2008.

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

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

[25] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proc. ACM ASPLOS*, 2002.

[26] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *Proc. USENIX OSDI*, 2008.

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