New Container Architectures for Mobile, Drone, and Cloud Computing

Alexander Van't Hof

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2023

Abstract

New Container Architectures for Mobile, Drone, and Cloud Computing

Alexander Van't Hof

Containers are increasingly used across many different types of computing to isolate and control apps while efficiently sharing computing resources. By using lightweight operating system virtualization, they can provide apps with a virtual computing abstraction while imposing minimal hardware requirements and a small footprint. My thesis is that new container architectures can provide additional functionality, better resource utilization, and stronger security for mobile, drone, and cloud computing. To demonstrate this, we introduce three new container architectures that enable new mobile app migration functionality, a new notion of virtual drones and efficient utilization of drone hardware, and stronger security for cloud computing by protecting containers against untrusted operating systems.

First, we introduce Flux to support multi-surface apps, apps that seamlessly run across multiple user devices, through app migration. Flux introduces two key mechanisms to overcome device heterogeneity and residual dependencies associated with app migration to enable app migration. *Selective Record/Adaptive Replay* to record just those device-agnostic app calls that lead to the generation of app-specific device-dependent state in services and replay them on the target. *Checkpoint/Restore in Android (CRIA)* to transition an app into a state in which device-specific information the app contains can be safely discarded before checkpointing and restoring the app within a containerized environment on the new device.

Second, we introduce AnDrone, a drone-as-a-service solution that makes drones accessible in

the cloud. AnDrone provides a drone virtualization architecture to leverage the fact that computational costs are cheap compared to the operational and energy costs of putting a drone in the air. This enables multiple virtual drones to run simultaneously on the same physical drone at very little additional cost. To enable multiple virtual drones to run in an isolated and secure manner, each virtual drone runs its own containerized operating system instance. AnDrone introduces a new *device container* architecture, providing virtual drones with secure access to a full range of drone hardware devices, including sensors such as cameras and geofenced flight control.

Finally, we introduce BlackBox, a new container architecture that provides fine-grain protection of application data confidentiality and integrity without the need to trust the operating system. BlackBox introduces a *container security monitor*, a small trusted computing base that creates separate and independent physical address spaces for each container, such that there is no direct information flow from container to operating system or other container physical address spaces. Containerized apps do not need to be modified, can still make full use of operating system services via system calls, yet their CPU and memory state are isolated and protected from other containers and the operating system.

Table of Contents

List of Figures

List of Tables

Acknowledgements

My research and this dissertation would not have been possible without the help and support of my colleagues, friends, and family. I want to thank my advisor, Jason Nieh, for his years of guidance, patience, and support. I also want to thank the rest of my dissertation defense committee, Roxana Geambasu, Junfeng Yang, Ronghui Gu, and Jeremy Andrus for their time and helpful feedback. Naser AlDuaij and Shih-Wei Li have been great friends and office mates and were always there to provide valuable insight or much-needed levity. I am grateful to my parents and sister for the sacrifices they have made for me. My research was supported in part by a Google Research Award, an IBM PhD Fellowship, OPPO, a Guggenheim Fellowship, DARPA contract N66001-21-C-4018, and NSF grants CNS-1162447, CNS-1422909, CCF-1162021, CNS-1717801, CNS-1563555, CCF-1918400, CNS-2052947, CCF-2124080.

Chapter 1

Introduction

Virtual machines (VMs) have long been used to share computing resources securely [1]. VMs are virtual computers with full operating system (OS) instances that can run simultaneously on a single machine. The OS within a VM behaves the same as it would if it had complete control of physical hardware but only controls virtualized hardware. Most commonly, VMs are supported by software running at a higher privilege level than an OS, known as a hypervisor [2, 3, 4]. Hypervisors manage and run VMs while isolating them from each other and providing them with virtualized hardware. Although VMs allow for securely multiplexing shared hardware, they are resource intensive since each VM runs its own OS instance, resulting in the shared hardware needing to run potentially many OS instances.

To provide isolated virtual computers on shared hardware without running multiple OS instances, containers were created [5, 6, 7]. Containers are a lighter-weight alternative that run together directly on a single OS and are isolated from each other through the use of resource namespaces [8] supported at the OS-level. These resource namespaces separate and isolate system resources while providing unique resource identifiers, such as process IDs (PIDs). For example, processes running in a container run within their own process namespace and have PIDs that are unique within that container. One container's PID 1 process is not the same as another container's PID 1 process, and outside of the containers, neither process appears as PID 1 to the OS and uncontainerized apps. Processes running in a container cannot see processes not belonging to that container's process namespace. Similar namespaces are used for system resources such as mounted file systems, inter-process communication (IPC), networking, and user IDs.

Containers include everything an app needs to run within its virtualized environment, including its own filesystem, shared libraries, and binaries, but do not need to contain an OS itself. This content is packaged together as a *container image*. A user can then download and run a container image using a *container manager*, like Docker [9]. To create a container, the container manager unpacks a container image, mounts the filesystem within, and then runs the app within the confines of a configurable amount of resource namespaces that comprise the conceptual container. The processes then run within this container, directly on the OS, without any virtualized hardware overhead. Since the container image provides all necessary dependencies, it will run reliably on any supported platform regardless of the host OS environment. Additionally, when running in the cloud, containers are able to be automatically provisioned, managed, and migrated by a *container orchestrator*, such as Kubernetes [10].

Since containers do not need an entire OS instance to support apps, they offer a low resource footprint, requiring only enough resources for the apps themselves. Since only the apps within a container need to be instantiated, they offer fast start-up times. By relying on the OS' native I/O facilities, containers do not need to introduce additional complex I/O virtualization support or hardware and offer minimal I/O performance overhead. Hardware requirements for containers are modest, and containers can even run on hardware without virtualization extensions. These advantages enable containers to run across a much broader array of devices than VMs.

While containers have mostly been used for cloud computing deployments just like VMs, this dissertation explores how we can take advantage of the ability of containers to run on almost any kind of device to enable new kinds of functionality. Some previous work has considered how containers can be used with smartphones, for example. Cells [11] enables multiple virtual phones on a single smartphone through the use of containers augmented with minor changes to the OS to multiplex device usage among the virtual phones. Unlike VMs, apps in containers interact directly with the software providing its virtualized environment. This allows Cells to utilize existing device drivers and other code instead of having to build support and features from scratch as VMs would require. Additionally, with VMs, the hypervisor has very little insight into what an app is doing,

but with containers, the OS is heavily involved with the app as it runs. This tighter coupling of the virtualization layer and virtualized environment, together with the virtualization layer having greater insight into the virtualized environment, can enable containers to offer new functionality across a range of platforms.

Beyond enabling new functionality, this dissertation also explores how we can make containers more secure. Although containers running on shared hardware offer many benefits over VMs, a significant downside is an increased security risk. Apps in VMs run almost completely isolated from other VMs, often only sharing a small hypervisor with a narrow application binary interface (ABI). However, apps in containers all run on the same large commodity OS with a wide system call interface. If an app in one container can exploit a vulnerability in the OS, it can gain access to container data, compromising the confidentiality and integrity of all containers on the system. Given the enormous and complex code bases of OSes, it is inherently difficult to trust that an OS doesn't contain a vulnerability that puts containers at risk.

My thesis is that new container architectures can provide additional functionality, better resource utilization, and stronger security for mobile, drone, and cloud computing. To demonstrate this, we introduce three systems with novel container architectures enabling mobile app migration, drone hardware sharing, and app security against an untrusted OS.

First, we present Flux [12] to support *multi-surface* mobile apps, apps that run seamlessly across multiple user devices allowing users to switch among their devices. Flux achieves multi-surface computing through app migration. Flux enables any app to migrate directly from one device to another. Devices can be different smartphone and tablet hardware, and Flux ensures that once an application is migrated to a guest device, it is able to make full use of the guest device's hardware, including resizing and reformatting the application content to fit the display of the guest device. Flux accomplishes this seamless app migration across heterogeneous devices by introducing two novel mechanisms: *Selective Record/Adaptive Replay* and *Checkpoint/Restore In Android (CRIA)*. Selective Record/Adaptive Replay interposes on app calls to device and system services to eliminate residual dependencies by *selectively* recording calls that modify device state.

During resume, these calls are then *adaptively* replayed to accurately recreate this device state on the target. CRIA checkpoints the critical user- and OS-level state of the running app at the source and restores it at the target within a containerized environment. Combining these two mechanisms, Flux introduces a container-based migration architecture with support for mobile apps across heterogeneous devices.

Second, we present AnDrone [13], a drone-as-a-service solution making drones accessible in the cloud to interested third parties. AnDrone provides users with lightweight *virtual drones* that can be configured in the cloud with various apps and services of interest to a user, then safely deployed and multiplexed on real drone hardware. AnDrone takes advantage of the observation that computational costs on drones are cheap compared to the operational and energy costs of putting a drone in the air, making it very efficient to multiplex multiple virtual drones on a physical drone to maximize the utility of drone flight time. Each virtual drone has its own containerized operating system environment with which to run its tasks and provides online interactive access to the drone during flight. AnDrone controls physical device access to isolate virtual drones from one another and preserve drone hardware safety. AnDrone achieves this by introducing a novel *device container* architecture with location-based control. The device container architecture isolates virtual drones from each other and from devices, transparently decoupling apps from low-level device implementation and interfaces.

Finally, we present BlackBox [14], a new container architecture that provides fine-grain protection of application data confidentiality and integrity without the need to trust the OS. BlackBox introduces a *container security monitor (CSM)*, a new mechanism that leverages existing hardware features to enforce container security guarantees in a small trusted computing base (TCB) in lieu of the OS. The monitor creates separate and independent protected physical address spaces (PPASes) for each container to enforce physical memory access controls but provides no virtualization of hardware resources. Physical memory mapped to one container's PPAS is not accessible in any other, providing physical memory isolation among containers and the OS. Since container private data in physical memory only resides on pages mapped to its own PPAS, its confidentiality and

integrity is protected from the OS and other containers. The CSM repurposes existing hardware virtualization support to run at a higher privilege level so that it can control nested page tables to create PPASes, but is itself not a hypervisor and does not do any virtualization. Instead, the OS continues to access devices directly and remains responsible for allocating resources. This enables the CSM to be minimalistic and simple while remaining performant. In addition, by supporting containers directly without virtualization, no OS or runtime code must be run within the secure environment enabling a minimal TCB to run within the container itself.

The contributions of this dissertation include:

1. The design, implementation, and evaluation of Flux, a system enabling apps to become multi-surface through app migration. Flux achieves this by extending and augmenting existing systems-level container migration to create a new architecture capable of mobile app migration across heterogeneous devices.

2. Selective Record/Adaptive Replay, a mechanism to record just device-agnostic app calls that lead to the generation of app-specific device-dependent state in services and replay them on the target during migration.

3. Checkpoint/Restore in Android (CRIA) to transition an app into a state in which device-specific information the app contains can be safely discarded before checkpointing and restoring the app within a containerized environment.

4. The design, implementation, and evaluation of AnDrone, a drone-as-a-service solution providing users with virtual drones supported by a novel device container based architecture.

5. Virtual Drones, cloud-configurable lightweight containerized drone instances capable of sharing physical drone hardware during a flight.

6. A mechanism for selectively publishing Android services to different containerized Android instances enabling the sharing of physical hardware among multiple Android instances.

7. The design, implementation, and evaluation of BlackBox, a container architecture providing app data confidentiality and integrity on an untrusted OS with a minimal TCB.

8. Container Security Monitor, a mechanism to create separate and independent protected physical addresses spaces for container instances.

This dissertation is organized as follows. Chapter 2 presents Flux, a system for supporting multi-surface apps. Chapter 3 presents AnDrone, a drone-as-a-service solution for maximizing flight usage. Chapter 4 presents BlackBox, a new container security architecture ensuring app data confidentiality and integrity. Finally, Chapter 5 presents conclusions and future work.

Chapter 2: Flux: Multi-Surface Computing in Android

2.1 Introduction

Users increasingly own multiple mobile devices of various shapes and sizes, with a recent survey reporting an average of roughly three devices per person [15]. Accordingly, there is a trend to run applications (apps) on multiple devices or *surfaces*. For example, it is possible to begin a movie using the Netflix app on a phone and switch to a larger screen to continue watching. In general, we expect to see more *multi-surface* apps emerge, including (1) switching from a larger device to a smartphone to travel, (2) displaying from a mobile device to a projector, (3) switching to a different device when the battery is running low, or even (4) collaboratively using an app during meetings, allowing multiple people to view, modify, and contribute.

However, despite this growth, there is little system support for multi-surface apps. Today, there are two trending approaches. The first approach is *screencasting*, in which screen output from one device is sent to another [16, 17, 18, 19, 20, 21, 22, 23]. For example, Apple AirPlay [24] allows content on an iOS device to be displayed on an Apple TV. However, the app continues to run on the original device, still limited by its computing power and battery life. It cannot take advantage of the capabilities of the new device, such as CPU, GPU, or memory. Furthermore, apps often need to be explicitly written for systems such as AirPlay to achieve the best user experience. The second approach is to use a *cloud-based* approach in which the actual app content is stored in a back-end in the cloud. For example, iCloud or Google Drive and devices like Chromecast [25] make cloud back-ends pervasive. However, cloud-based approaches suffer from both a dependence on connectivity and a growing distrust of cloud providers to handle data. Cloud provider distrust is actually prohibitive in many enterprise environments with sensitive client data [26, 27].

We propose a third approach to achieve multi-surface computing: app migration. App migra-

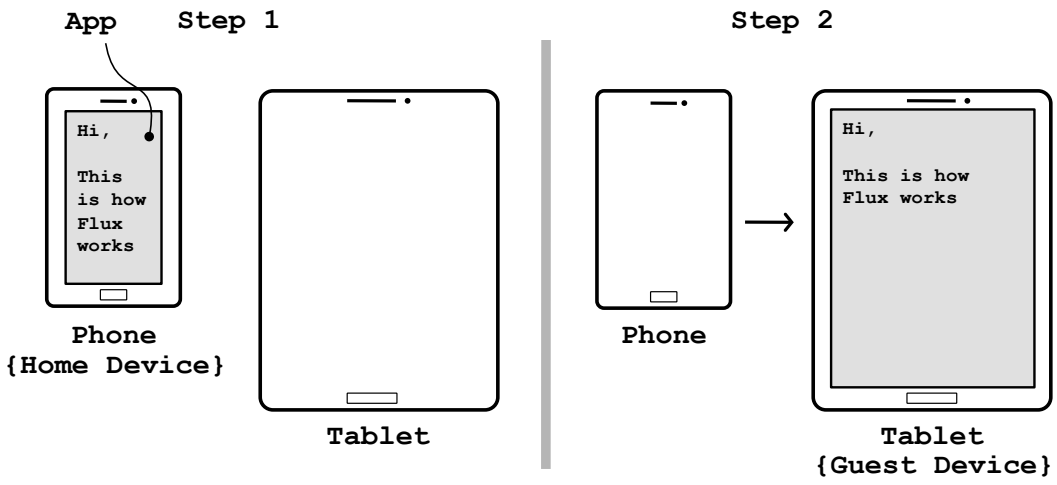

Figure 2.1: Multi-surface support through app migration: swipe to migrate unmodified app between paired devices without cloud support.

tion enables the app to take advantage of the device in use, allows the original device to be used for other tasks, and does not require connectivity to a cloud provider; if disconnected from the Internet, devices can use ad-hoc networking. Furthermore, because it is implemented at the systems level, apps do not need to be written to support multi-surface operation.

Migration of an app is non-trivial. For example, in Android, even though apps are written expecting to be killed at any moment due to memory pressure, many apps do not automatically save all of their runtime state. If these apps crash, the state is lost. Therefore, it is not possible to migrate an app by simply killing it and starting it on the destination.

Furthermore, app migration between mobile devices is more complicated than many other environments due to device heterogeneity. Smartphones and tablets are tightly integrated hardware platforms that come in many different sizes and incorporate a plethora of devices using non-standard interfaces, such as GPUs and cameras. As of 2014, the OpenSignal database shows 18,796 different Android devices, up from 11,868 reported a year earlier [28].

Device heterogeneity complicates the usual challenges of *residual dependencies*, or state left in the source system after migration, in two ways. First, apps interact with *system services*, shared processes that may maintain app-specific and device-specific state. It is not feasible to migrate a shared system service along with the app or extract the app-specific state from the service. Even

if the entire state of the system services was saved and restored on the target device, it may not work because the services manage device-specific state. Second, the running apps themselves contain—potentially device-dependent—state that is not easily accessible to the system. Blindly saving device-dependent app state and restoring it would not work across the thousands of different Android devices.

To address these problems, we introduce Flux, an Android framework for app migration. As shown in Figure 2.1, Flux enables any app to migrate directly from one device to another without any cloud support. Devices can be different smartphone and tablet hardware, and Flux ensures that once an application is migrated to a guest device, it is able to make full use of the guest device's hardware, including resizing and reformatting the application content to fit the display of the guest device. Flux accomplishes this seamless application migration across heterogeneous devices by introducing two novel mechanisms: *Selective Record/Adaptive Replay* and *Checkpoint/Restore In Android (CRIA)*.

Selective Record/Adaptive Replay eliminates residual dependencies due to system services. Specifically, during app execution, Flux interposes on app calls to system services and only records those that modify app-specific device state, automatically discarding stale interactions. Selective Record is also used to guarantee correctness of Android services after migration. During resume, the recorded app calls are *adaptively replayed* through Flux's service contextualization proxy to match the guest OS's system services. Importantly, this record/replay mechanism ensures that device-dependent state in the source is accurately recreated on the target.

CRIA checkpoints critical user- and OS-level state of the running app at the source and restores it at the target. A key feature of CRIA is that it integrates with Android to eliminate most residual dependencies on the system and customize the restoration of checkpointed state in a manner tailored to the target, supporting device heterogeneity. CRIA deals with device-specific state by putting the app into such a state that it discards much of the device-specific state on the source. Next, CRIA checkpoints core app state, including app-specific state in Android specific drivers such as *Binder*, the IPC mechanism through which apps interact with the system-provided services

that front most devices, e.g., GPS and camera. On restore, Flux leverages Android app initialization mechanisms to inform the app of changes to hardware state so that app-specific device state can be reconstructed in a manner customized to the guest platform, including matching the UI to the screen size of the guest platform. Restoring checkpointed state reestablishes the app's Binder connections to system services, now at the target.

We have implemented and evaluated a working prototype of Flux on Android. Our results show that Flux successfully migrates a wide range of the top apps from the Google Play store across different smartphone and tablet hardware running different OS kernel versions. We show that the runtime overhead of Flux during app execution is negligible. Not surprisingly, the migration time is dominated by network transfer times. Nonetheless, we found that migration time and the amount of state transferred was modest in most cases, demonstrating that Flux is fast enough for interactive use.

This paper presents the design and implementation of Flux. Section 2.2 gives an overview of Android. Section 2.3 describes the Flux architecture, focusing on Selective Record/Adaptive Replay and CRIA. Section 2.4 presents experimental results. Section 2.5 discusses related work. Finally, we present some concluding remarks and directions for future work.

2.2 Android Background

Figure 2.2 gives an abridged overview of the Android system components that apps are dependent on and are therefore critical during app migration. An app in Android is written in Java and runs inside an isolated instance of the *Dalvik* VM. Typically, an app runs in a single process; less commonly, an app may be split into multiple processes. Apps are installed with the PackageManagerService, which tracks app metadata such as requested permissions. An app is typically isolated to a single data directory through filesystem permissions and access to storage such as an SD card requires explicit permission upon installation. An app consists of any number of *activities* and, when necessary, talks to *services* via Binder, Android's primary IPC mechanism. An activity is an app component providing a UI with which users can interact with to perform tasks, such as send an

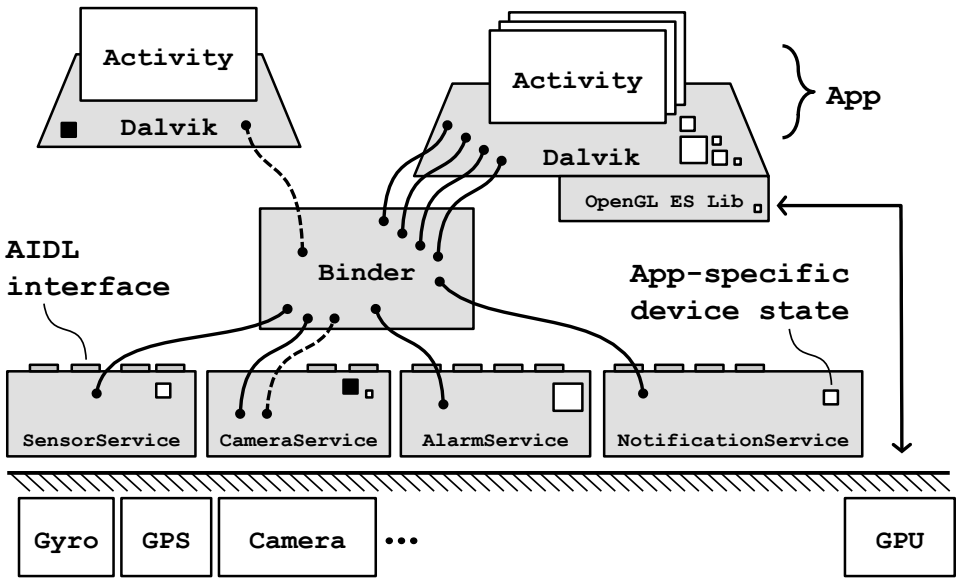

Figure 2.2: Overview of Android system components and their relation to one another through Binder.

email, or dial the phone. A service is an app or system component that can perform long-running operations in the background without a UI. When migrating, the various device state associated with both activities and services must be correctly handled.

Android apps rely heavily on interactions with shared, long-running system services. For example, the NotificationManagerService allows apps to post notifications to the status bar, and the AlarmManagerService allows apps to schedule code to be run at some point in the future. Apps communicate with these services and with each other exclusively via Binder, either explicitly via RPC service interfaces or through *Intents*. Intents are messaging objects used to request an action from another app, which can be broadcast to all relevant apps by the ActivityManagerService. To simplify the creation of RPC service interfaces, the Android Interface Definition Language (AIDL) allows programmers to write an interface by simply defining method prototypes. AIDL will then generate the necessary serialization and IPC code required for the interface.

In addition to distributing Intents, the ActivityManagerService is responsible for managing the running of Android applications, including starting and stopping app components, and registering app-requested BroadcastReceivers, which act as listeners for apps for various events, e.g., informing them of WiFi status changes. Another duty of the ActivityManagerService is controlling the

life cycle of activities. In Android, activities transition between various states of their life cycle. After creation, an activity enters the Resumed state, where it remains until it is sent to the background or another activity partially obscures it from view. Once sent to the background, the activity transitions to the Paused state. In this state, the activity no longer receives user input and cannot execute any code. If the activity is not quickly brought back to the foreground, the Android task idler will place it into the Stopped state. In this state, the activity is guaranteed to not be visible to the user and it will no longer be able to render its user interface.

The user interface of an Android app consists of a Window, provided by the WindowManagerService, for each activity. A Window, similar to a desktop window, contains a single Surface in which the content of the Window is rendered. This Surface will be destroyed when the app is in the Stopped state to conserve resources. Each Window also has a View hierarchy attached to it. View hierarchies are rooted by a ViewRoot and consist of ViewGroups containing Views, which are interactive UI elements. Each time a Window is to be rendered, the View hierarchy is traversed and each View draws its portion of the UI.

Communication with devices takes place via system-provided Binder services, e.g., the SensorService. An exception is the GPU, which is interacted with directly using the standardized OpenGL ES library that abstracts away hardware details. Similar to other hardware libraries in Android, OpenGL consists of both a generic library, presenting apps with a well-known API, and a vendor-specific library, implementing device-specific code called by the generic library.

All Android apps rely on the Android version of the Linux kernel: it is therefore a shared resource. In the kernel, Binder is implemented as a driver. Binder communication typically consists of clients talking to services. In Binder, the service side is dubbed a *node* and all clients reference nodes via process-specific *handles*, identified by a simple integer. Communication to another Binder node cannot occur without first being given a reference to it by the process who created it or a process already holding a reference to it. Therefore, services wishing to offer other processes an RPC interface must register themselves with the userspace *ServiceManager*. The ServiceManager maintains a registry of Binder references corresponding to names given when the service was

registered. It is up to the service itself to decide whether or not a calling process has permission to make a particular RPC. Other features of the kernel include *ashmem*, a shared memory driver; *pmem*, a physically contiguous memory allocator used by devices like the GPU; an alarm driver, allowing the AlarmManagerService to schedule alarms that can trigger regardless of the machine's sleep state; *wakelocks*, a power management feature used to keep the machine awake while a wake-lock is held and to sleep otherwise; and the Logger driver. When migrating between devices, the state of all these Android specific drivers must be considered.

2.3 Flux

We assume an environment that consists of many mobile devices running Flux. An app can be installed on some, but not necessarily all, mobile devices. The device on which the app is natively installed is called the *home* device. As shown in Figure 2.1, users in our environment can migrate any running app, along with all its active state, from its home device to other *guest* devices. We do not rely on any back-end (cloud) support or modifications to the app.

2.3.1 Migration Life Cycle

Figure 2.3 shows that Flux consists of a number of components, which are highlighted in gray. We describe the high level role of each component in the context of a migratable app's life cycle, as depicted in Figure 2.4: *Pairing*, *App Execution*, *Migration Out*, and *Migration In*.

Pairing. Before a user migrates an app from the home device to a guest device, Flux performs a one-time pairing operation that synchronizes the home device's core framework and libraries to a custom location on the guest's data partition. This is needed because the core framework and library binaries may differ across devices; the frameworks and libraries used by an app must remain the same before and after migration. The differences between these files are generally small. Consequently, the synchronization operation is performed efficiently since most files are linked against the identical files on the guest's system partition. In our current implementation, we use `rsync` for synchronizing files and its `--link-dest` option for linking identical files.

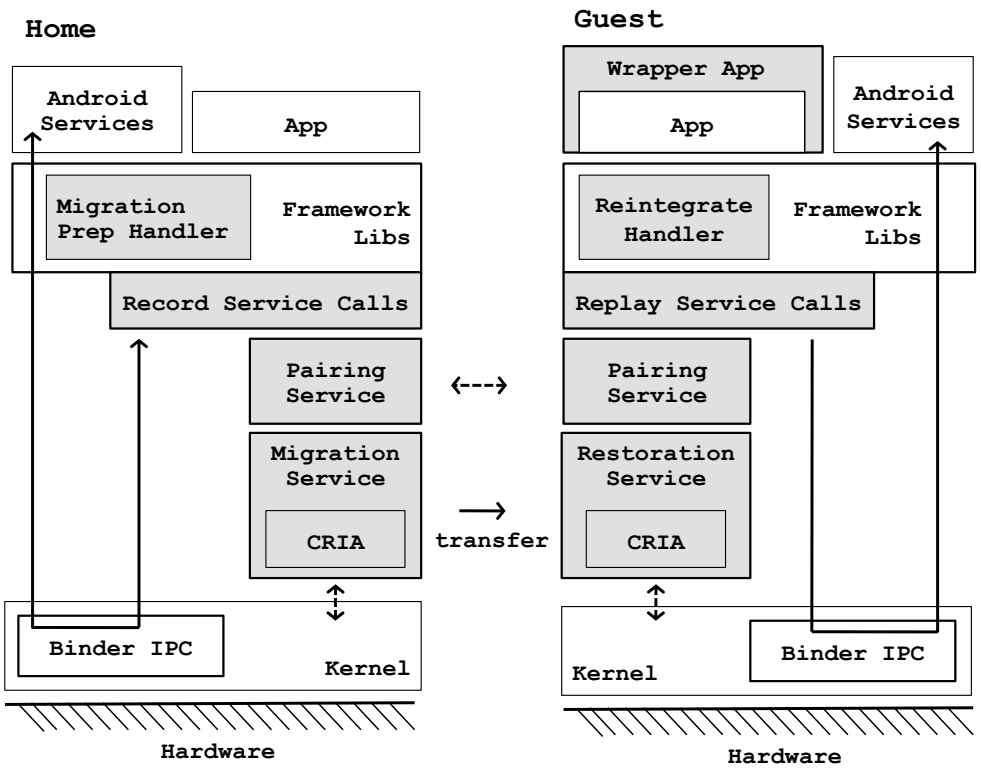

Figure 2.3: Overview of Android components involved in migration. Components added by Flux highlighted in gray.

Similarly, Flux also verifies and synchronizes the home device's app binaries, known as Android Package Files (APKs), and app data files to the guest device. This includes any app-specific data directories residing on the SD card, but not general SD card data available to all apps with SD card access. Since apps may be updated frequently, the paired APK is verified prior to migration and updated if necessary.

As part of the pairing, Flux pseudo-installs the APK's metadata on the guest with its PackageManagerService. This allows the guest to be aware of the app's permissions and components but does not actually install the app data, such as the app executable and other resources. This pseudo-installed app acts as a wrapper when migrating in; additionally, it differentiates a migrated app on the guest device from the natively-installed version.

Due to the fragmentation of the Android market, app binaries are typically designed to run across a wide range of Android versions. However, if a particular APK requires an API level that

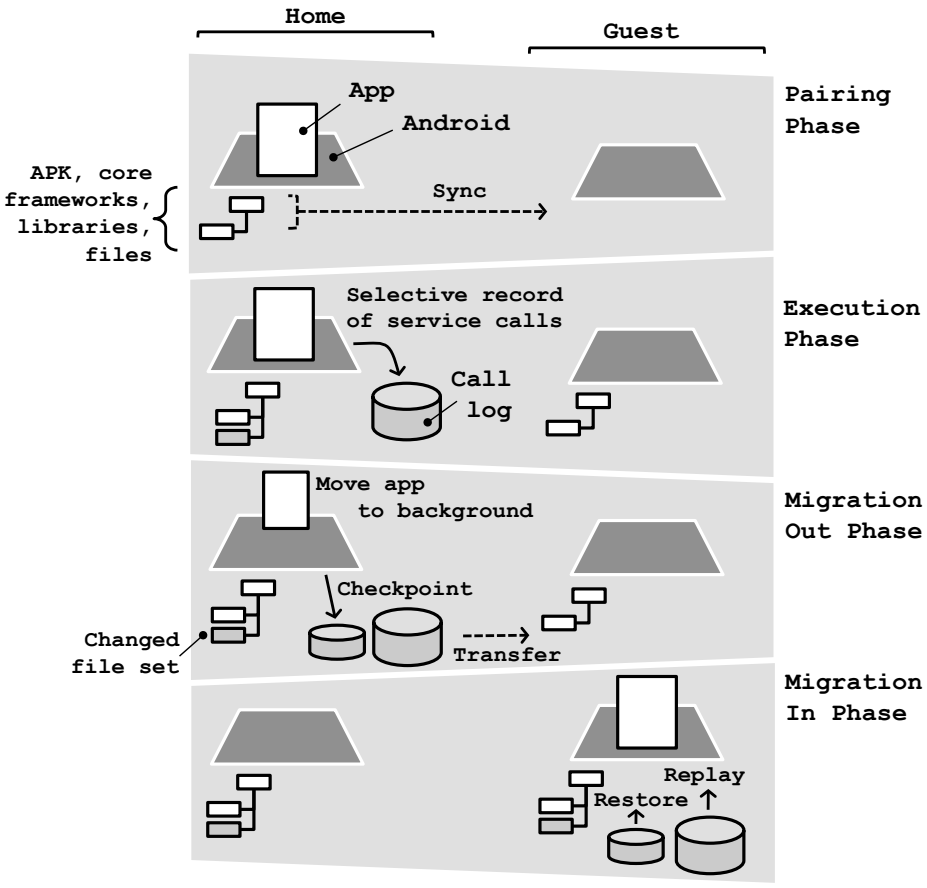

Figure 2.4: Workflow phases of Flux migration.

is incompatible with the software stack of the guest device, Flux informs the user the app cannot be migrated.

App Execution. During app execution, Flux selectively records an app's interactions with system services through Binder's IPC mechanism. This recording functionality, described in Section 2.3.2, uses framework-level *decorators* of the system services' RPC interface. Additionally, the recording functionality is provided in core framework-supplied libraries and is transparent to the app. The recorded log is primarily used to restore the app-specific state of system services once the app has migrated to a guest device, avoiding the need to migrate these services along with the app. It is kept small by automatically discarding stale calls.

Migration Out. A user initiates a migration operation through a two-finger vertical swiping gesture. Flux's first step is to use Android's built-in mechanisms to free as much device-specific state

as possible. Specifically, Flux instructs apps to go to the background, which helps free drawing surfaces. Then, Flux triggers a low-memory condition, which further releases graphic-related resources. Finally, Flux extends OpenGL to remove any remaining vendor-library-specific state.

Next, Flux checkpoints the app's process(es). Because the primary way in which Android apps interact with the rest of the system is through Android's Binder IPC mechanism, Binder IPC state must be saved as part of the checkpoint. Flux achieves this using CRIA, as described in Section 2.3.3. Flux's checkpoint includes not only per-process app state, but also the recorded log of calls made by the app to interact with system services. Once complete, the checkpoint image is compressed and sent to the guest device, along with the app's data directory.

Migration In. To restore the app from the checkpoint on the guest device, Flux uses the wrapper app created at pairing time as a shell in which to restore the checkpointed image of the migrated app. The wrapper app is launched in a private virtual namespace for process identifiers to ensure that app processes see the same identifiers even if the underlying operating system identifiers may have changed. The wrapper app is also jailed to the previously synchronized filesystem containing the home device's libraries and the app's APK. Once complete, Flux restores the app from the checkpointed image, as discussed in Section 2.3.3, including re-establishing the same Binder state for the app so that the app sees the same Binder handles. To complete the integration of the app into the new guest environment, Flux informs the app of any changes to hardware, and replays the recorded service calls, as discussed in Section 2.3.2, for the guest device's services to restore necessary state on behalf of the app, permitting the app to interact with system services right where it left off.

Flux does not restore the app's original network state, but rather leverages the fact that mobile apps are typically built around transient wireless connectivity and therefore are designed to correctly handle connectivity changes. When restoring a migrated app, Flux informs it of a loss of connectivity and availability of a new connection, allowing the app to handle the connectivity interrupt in the normal way. Finally, the app is brought to the foreground so that it becomes visible and available for use by the user.

2.3.2 Selective Record/Adaptive Replay

Because system services are shared by many apps and may have device-specific implementations, it is not desirable to migrate the system services themselves. Instead the corresponding system services on the guest device should take over after the app has been migrated. One approach could be to add checkpoint and restore hooks to every service an app may interact with, thereby enabling the extraction and restoration of a service's app-specific state. Doing so would be an overwhelming undertaking, requiring specialized knowledge of service and device implementation details that vary from one device to another. Instead of checkpoint-restore, Flux introduces *Selective Record/Adaptive Replay*.

In a straightforward implementation of record-replay to migrate app-specific state, all calls that update the app-specific state in system services are recorded, then deterministically replayed on the guest device with its system services. This approach, however, has three key problems. First, system services support a wide range of features, each with different behavioral semantics. A one-size-fits-all approach to recording service calls will not suffice. For some services, particularly those that interact with the user, e.g., notification or alarm, simply recording and replaying all service calls would result in an incorrect state, e.g., the user would see past notifications that he has already acknowledged. Second, computing resources on a mobile device are scarce. A straightforward record-replay mechanism may unnecessarily record/replay all calls, wasting scarce mobile resources during app execution and migration and introducing an unacceptable latency waiting for the entire log to be replayed. Third, services across devices may not be identical and, in some cases, not available. A straightforward record-replay mechanism assumes a homogeneous environment and does not adapt to device variations.

To address these problems, Flux introduces Selective Record/Adaptive Replay to only record and replay calls to system services that are relevant to reproducing the current app-specific state in the respective services. As shown in Figure 2.5, since apps interact with system services via Android's Binder IPC mechanism, Selective Record simply interposes on the service interface calls used by Binder. These calls are based on standard APIs that are device independent, avoiding

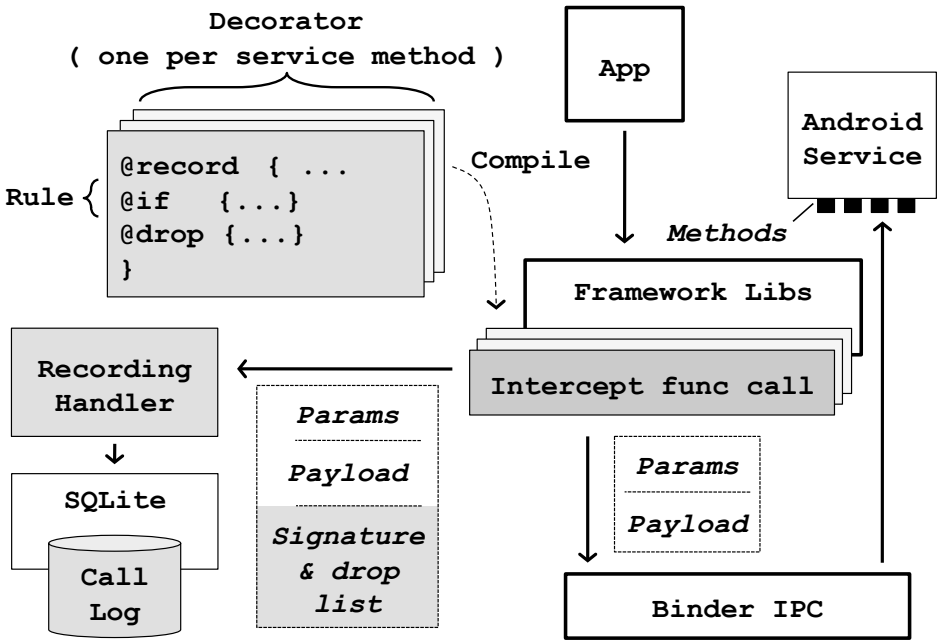

Figure 2.5: Selective Record and its place in the Android system. Flux-specific components highlighted in gray.

the need to understand device-dependent implementation details. Selective Record leverages the higher-level semantics available at these interfaces to identify which recorded calls are no longer relevant to the current app-specific state of a given service, and thereby can be discarded. Adaptive Replay then modifies the recorded calls in a way that best suits the characteristics of the guest device.

Selective Record. To capture the higher-level semantics from Android frameworks, Flux provides decorators that can be used by framework developers to instrument IPC service interface definitions. The decorators identify what calls should be recorded and how they affect the current state of the system. Our expectation is that the decorators are simple to use and require only minimal additions to existing frameworks. To further simplify the use of decorators, Flux takes advantage of interface definition languages (IDLs), which are commonly used to generate RPC interface serialization code. Flux extends the IDL to support decorators. Specifically, the Android IDL (AIDL), used for defining system service interfaces. For decorated interface methods, AIDL generates the necessary code to call our record function. The record function then asynchronously performs the

SYNTAX	PURPOSE
`@record`	Indicate that calls to this method should be recorded.
`@drop` [method name], ...	Remove all previous calls to this method.
`@if` [arg], ... `@elif` [arg], ...	Qualifies `@drop` to only remove previous calls if all args given match.
`@replayproxy` [method]	When replaying, call proxy [method] instead of replaying the actual call.
`this`	A keyword representing the current method being decorated.

Table 2.1: Flux decoration syntax.

actual recording and the necessary removal of stale calls which no longer affect the current state of services.

Table 2.1 lists the decorators that are supported by Flux. The syntax is modeled after Python's decorators, hence the name. Each decorator indicates what action should be taken with the subsequent call. There are four basic constructs. The `@record` statement indicates that calls to the respective function should be recorded to the log. The `@drop` statement indicates that previous calls to the respective function should be discarded from the log. The `@if` statement is used to qualify a `@drop` statement to only discard previously recorded calls from the log if all arguments provided with the `@if` statement match. Finally, the `@replayproxy` statement is used during replay to indicate that an alternative proxy method should be used instead of replaying the actual recorded call, thereby modifying the resulting replay.

Table 2.2 provides a full listing of all the decorated Android services along with the number of lines of code (LOC) required, separated into those that manage hardware devices and those that do not. For comparison purposes, it also shows the number of methods for each service interface, which provides a loose measure of the complexity of the respective interface. Generally speaking, services with larger interfaces require more lines of code to decorate. A few services are not yet decorated in the current Flux prototype, so their LOC are indicated as TBD. Most services require less than 50 LOC, except for ActivityService and AudioService, which require 130 and 150 LOC, respectively. As shown in Table 2.2, these two services also have larger interfaces than other

HARDWARE SERVICE	METHODS	LOC
AudioService	71	150
BluetoothService	202	TBD
CameraManagerService	8	31
ConnectivityManagerService	59	26
CountryDetectorService	3	5
InputMethodManagerService	29	37
InputManagerService	15	11
LocationManagerService	13	15
PowerManagerService	19	14
SensorService	6	94
SerialService	2	TBD
UsbService	19	TBD
VibratorService	4	26
WifiService	47	54
SOFTWARE SERVICE	METHODS	LOC
ActivityManagerService	178	130
AlarmManagerService	4	20
ClipboardService	7	6
KeyguardService	22	16
NotificationManagerService	14	34
NsdService	2	3
TextServicesManagerService	9	16
UiModeManagerService	5	9

Table 2.2: Decorated services in Android comparing the number of methods for each service interface and the number of lines of Flux decorator code for the service.

services.

Example: NotificationManager. The NotificationManager, the AIDL interface for the NotificationManagerService, is used by apps to post and maintain notifications displayed on the status bar and in the notification drawer. It provides a simple example of how selective recording is performed. To migrate the app state, we must record these notifications to the guest device along with the app. Figure 2.6 shows a portion of an IDL defined interface derived from Android's actual NotificationManager, and Figure 2.7 shows the same definition with Flux decorations. The `@record` statement above `enqueueNotification` indicates that all calls to this function should be recorded. Inside the `@record` block above `cancelNotification`, the `@if` statement indicates that the n-tuple `(id,)` will be used as a signature to determine if a call to `can-`

```
interface INotificationManager {
    void enqueueNotification(int id,
              Notification notification);
    void cancelNotification(int id);
}
```

Figure 2.6: Simplified interface definition for NotificationManager.

```
interface INotificationManager {
    @record
    void enqueueNotification(int id,
                    Notification notification);

    @record {
        @drop this, enqueueNotification;
        @if id;
    }
    void cancelNotification(int id);
}
```

Figure 2.7: Simplified interface definition for NotificationManager with Flux decorations.

`celNotification` matches the signature of any previous calls to methods in the `@drop` list. The `@drop` statement contains a list of interface methods whose effect on the device state will no longer matter if `cancelNotification` is called with a matching signature. If a signature matches, any matching previous calls will be removed from the record. `this` is a keyword in the drop list, indicating that the call to the decorated method `cancelNotification`, should not be recorded if there is a match. Note that, because of the simplicity of this example, the decorations comprise a substantial portion of the resulting lines of code of the interface. However, this represents a small percentage of the total number of lines of code generated by AIDL to implement the interface, and it also represents an even smaller percentage of the total number of lines of code to implement the actual service.

Adaptive Replay. Once an app has been migrated to a new device, changes to hardware state that can normally be modified by the user, such as the WiFi state, are replayed to any listeners the app has set up. Should the guest device not contain hardware that was previously in use, e.g., GPS, the user is given the option to allow communication with that device to continue to take place over the network.

To alter the replay as needed, the `@replayproxy` statement may be used to decorate service methods to indicate that when a particular method is called during replay, an alternative proxy method should be used instead. For example, a proxy method could be used to adjust volume levels of music being played in accordance with the relative volume level differences between the home and guest devices. This approach is specifically used to support services like the AlarmManagerService.

Example: AlarmManager. The AlarmManager, the AIDL interface for the AlarmManagerService, is used by apps to schedule tasks to be run at some point in the future. It provides an example of how an alternative proxy method is used on replay. In this case, knowing only the arguments to methods is insufficient for deciding which calls must be replayed. This is because alarms are set through an API call, but then typically expire with time, not by being explicitly removed through a subsequent API call. To set an alarm, an app calls the AlarmManager's `set` API method, specifying a time for the alarm to go off and an Intent to be broadcast at that time. The app will have registered a BroadcastReceiver to listen for this Intent in order to accomplish whatever task the alarm was set for. If the app is not currently running when the alarm expires, it will be started prior to the Intent broadcast. To prematurely cancel an alarm, an app can call the `remove` API method, specifying the Intent previously passed to `set`. When migrating an app we must also migrate any previously set, and still active, alarms. Figure 2.8 shows a portion of an IDL defined interface derived from Android's actual AlarmManager, and Figure 2.9 shows the same definition with Flux decorations. The decorations indicate that calls with the same *operation* argument to `set` and `remove` should be dropped from the record as either the alarm has been removed or replaced with a new alarm and the previous calls are no longer necessary or valid. However, if an alarm is set and not removed but triggered by the advancement of time, it is important to detect that the alarm has already been triggered and should not be triggered again. To handle this common case, the `@replayproxy` statement is used to indicate that when replaying calls, our `alarmMgrSet` method should be called instead of simply replaying the call. This method, as shown in Figure 2.10, will first verify if the alarm is still active and, if it is, replay the call using Java Reflection. The method

```
interface IAlarmManager {
  void set(int type, long triggerAtTime,
           in PendingIntent operation);
  void remove(in PendingIntent operation);
}
```

Figure 2.8: Simplified interface definition for AlarmManager.

```
interface IAlarmManager {
  @record {
    @drop this;
    @if operation;
    @replayproxy \
        flux.recordreplay.Proxies.alarmMgrSet;
  }
  void set(int type, long triggerAtTime,
           in PendingIntent operation);

  @record {
    @drop this;
    @if operation;
  }
  void remove(in PendingIntent operation);
}
```

Figure 2.9: Simplified interface definition for AlarmManager with Flux decorations.

compares against the time of checkpoint rather than the current time to avoid missing an alarm set to trigger while the app was mid-migration. This ensures that an alarm that is set for after the time of checkpoint will be triggered as intended after migration.

Example: SensorService. The SensorService is used by apps to receive events from sensors, e.g. accelerometers, gyroscopes, etc. It provides another example of using alternative proxy methods on replay. In this case, API calls return handles to objects, such as Binder objects and socket descriptors, which are used by apps; these return values are uncommon in app-facing Android system services. To receive sensor events, an app asks the SensorService for a SensorEventConnection via its `createSensorEventConnection` method. The SensorEventConnection is a Binder object with an interface of its own that allows the app to enable desired sensors and receive a Unix domain socket via a call to `getSensorChannel`, over which it will receive the sensor events on via the SensorService.

```
void alarmMgrSet(Class alarmMgrClass,
                 Object newAlarmMgr,
                 String method, int type,
                 long triggerAtTime,
                 PendingIntent operation) {
    if (triggerAtTime <= checkpointTime)
        return;

    Method set = alarmMgrClass.getMethod("set");
    set.invoke(newAlarmMgr, type,
                triggerAtTime, operation);
}
```

Figure 2.10: Simplified proxy method for replaying IAlarmManager.set().

When replaying calls to the SensorService, SensorEventConnection objects must be restored. This requires that the calls return the same handles to SensorEventConnection objects that the app was using before migration to ensure that the app continues to function properly after migration. Specifically, the Binder handle representing a SensorEventConnection and its respective Unix domain socket descriptor should remain the same after migration. To do this for the Binder object, a `@replayproxy` method is created for replaying the `createSensorEventConnection` call. The arguments supplied to this proxy method include the return value of the recorded call (the Binder handle representing a SensorEventConnection). This allows the proxy method to call the new device's SensorService to receive a new SensorEventConnection and map it to the correct Binder handle. Previously recorded calls to the SensorEventConnection will then be replayed. Similarly, to maintain the same descriptor for the Unix domain socket, a `@replayproxy` method is created for the SensorEventConnection's `getSensorChannel` call. This proxy will make the same call to the new SensorEventConnection's getSensorChannel method, obtaining a new connection with the SensorService (and by extension the Sensor). It will then `dup2` this descriptor into the original socket descriptor, reserved during restoration of the app.

Table 2.2 shows that although there are only 6 methods for the SensorService, it requires over 90 lines of code to decorate. The extra complexity here is due to the fact that this service is written natively in C++ and AIDL does not support generation of native code. The record/replay code that would normally be generated automatically through Flux's decoration syntax must be written by

hand, requiring more care and time than would otherwise be needed. In the future, AIDL can be extended to support generating native C++ code [29].

2.3.3 Checkpoint/Restore In Android (CRIA)

To support migration of an app's processes, Flux extends traditional checkpoint-restart mechanisms [30, 31, 32, 33, 5] in a manner that leverages the characteristics of Android to save the core state of the app on one device and restore it on another; we call this Checkpoint-Restore In Android (CRIA). There are four types of app state to consider for checkpointing: process, device, filesystem, and network state. As discussed in Section 2.3.1, filesystem state is synced across devices and network state is simply re-established on the guest device after migration so that it appears simply as a loss of connectivity to apps, which are expected to handle such interruptions on mobile devices. We focus here on checkpointing process and device state.

Process State. CRIA builds on the Checkpoint/Restore in Userspace (CRIU) project [30], which is supported in the mainline Linux kernel. Hooks in the kernel allow CRIU to transparently obtain and inject all necessary internal kernel state required to represent the state of a running process. As part of restarting the app after migration, the app is encapsulated in a private virtual namespace [5] to ensure that operating system resource identifiers such as process identifiers remain the same, even if the same numerical identifiers are already in use on the guest system.

CRIA extends CRIU to take into consideration Android-specific device drivers: Binder, Logger, ashmem, pmem, and wakelocks. Of these, Binder required the most support. As shown in Figure 2.11, to capture dependencies that result from the use of Binder, CRIA checkpoints and restores three types of Binder connections: (1) internal app, (2) external system services, and (3) external non-system services. App processes contain handles that refer to various Binder connections. CRIA checkpoints the Binder state of each app process, including Binder handles, references and buffers, and notes which references are internal versus external to system services, including recording the association between references to system services and those service names.

The restore process is different depending on the type of connection. For Binder connections

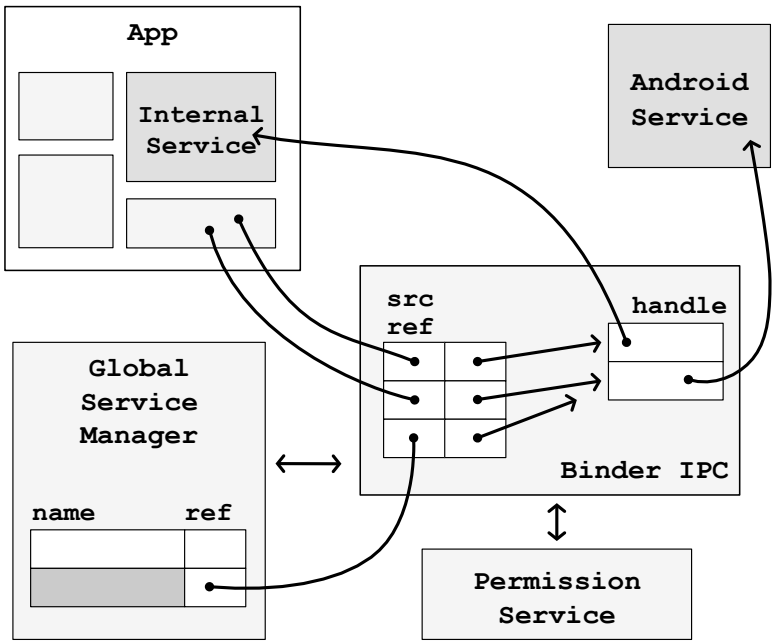

Figure 2.11: Binder dependencies captured with CRIA.

that are internal to the app, CRIA restores both ends of the connections. For Binder connections between the app and external system services, CRIA establishes new Binder connections with the same system services running on the guest device. CRIA asks the ServiceManager on the guest device for references to the equivalent new system services and injects those references in Binder with the previously issued handle identifier. For example, if the app references the NotificationManagerService using reference $id = 2$, it can continue to do so even on a new device with a different NotificationManagerService. This process only restores the connection between an app and various system services via Binder. As described in Section 2.3.2, app-specific state maintained by system services is restored via Selective Record/Adaptive Replay.

It is also possible that an app may have external Binder connections that connect to non-system services, such as non-system apps. A variety of solutions are possible to address this case, including migrating both connected apps or tethering the migrated app back to the home device. However, we have not encountered any such apps. For simplicity, CRIA currently checks for whether such Binder connections exist and if so, informs the user that the app cannot be migrated.

Support for the other Android-specific device drivers, Logger, ashmem, pmem, and wakelocks

was relatively straightforward. Adding support for the Android Logger driver required few changes since the device is used like any regular file and does not persist per-process state. Although direct support for ashmem is straightforward to implement, its use is limited. ashmem is primarily used by Dalvik to name memory regions. For the sake of simplicitly, we modified Dalvik to use `mmap` for obtaining memory instead of ashmem. After this, we did not encounter other instances of apps using ashmem at the time of checkpoint, so direct support for ashem was not needed. Similar to ashmem, CRIA support for pmem is not necessary due to freeing resources prior to checkpointing. Finally, CRIA support is not needed for wakelocks and alarms as these are only used by Android system services; therefore, their process-specific state is handled by Selective Record/Adaptive Replay.

Device State. Checkpointing device-specific state is especially difficult on mobile devices because of the lack of hardware standards in these vertically integrated platforms. In Android, there are two cases: (1) devices are used indirectly by apps via system services that manage those devices, and (2) devices are used directly by apps. As described in Section 2.3.2, Selective Record/Adaptive Replay addresses the migration of device state in the first case.

For the second case, the GPU is the only device used directly by Android apps. Migration of graphical context is difficult given the complexity of the hardware and software and the substantial amount of app and device-specific state involved. However, because using the GPU involves consuming substantial system resources, most mobile operating systems have support for dynamically removing and restoring GPU-related resources. CRIA leverages and extends this support to avoid the need to checkpoint and restore GPU-related state, dramatically simplifying the management of device state for migration. CRIA repurposes three types of Android mechanisms: background execution, low-memory condition, and conditional initialization.

CRIA leverages Android's background execution mechanism by instructing apps to revert to running in the background prior to being migrated. Because background apps are not visible to the user, various state associated with the visible interface of apps is not needed. By having an app run in the background, CRIA causes at least a partial removal of drawing surfaces and contexts

corresponding to the visible state of an app. However, other graphical hardware resources and OpenGL contexts will still be retained.

To eliminate the dependencies on the GPU hardware, CRIA leverages Android's low-memory mechanisms, which can force apps to free graphics-related resources. CRIA invokes a trim memory request for the migrating app with the highest severity level via Android's ActivityThread's `handleTrimMemory` method. `handleTrimMemory` requests that the WindowManager trim its memory via a `startTrimMemory` RPC method. This invokes the HardwareRenderer's `startTrimMemory` method causing its caches to be flushed, and then invokes all ViewRoots' `terminateHardwareResources` method. This then calls the HardwareRenderer's `destroy HardwareResources` and destroy methods causing all hardware rendering resources associated with those ViewRoots to be destroyed, the Canvas removed, and disables the renderer. ActivityThread will then call WindowManager's `endTrimMemory` method, which in-turn terminates all OpenGL contexts causing the HardwareRenderer to terminate and uninitialize OpenGL once all contexts are gone. The ViewRoot of the app is also destroyed, removing device-specific state that reference the ViewRoot.

Once completed, this leaves only a small amount of lingering native, graphics-related, vendor-library specific initialization state that must be removed. To do so, we extend Android's native OpenGL library with an `eglUnload` function. This is called after the HardwareRenderer is terminated and is used to completely unload the linked vendor-specific graphics libraries which are tied to the specific graphics hardware on the respective device, allowing for any new vendor-specific OpenGL library to be loaded when necessary.

Once an app is migrated and is being restored, CRIA leverages conditional initialization used by Android. Because Android is event-driven, various state used by apps is initialized on demand at time of use by checking first if the state is initialized before using it. CRIA reinitializes graphical context via the same initialization routines as used when starting an app. It takes advantage of conditional initialization to ensure that initialization is performed automatically due to the state of all objects appearing as if they were just created. Once graphics objects have been recreated and/or

initialized, all Views will be in an invalid state, forcing them to be redrawn as they were prior to migrating. An important benefit of this approach is that, because graphics state is reinitialized and redrawn on the guest device, the resulting device-specific state is customized for the guest device.

2.3.4 Discussion

In our design, we made several decisions to simplify the system's role in managing consistency between devices. At the same time, we considered the impacts of the diversity of the Android framework and "future-proofing" Flux against changing versions.

Native vs. Guest Apps. Our current design differentiates native apps from migrated apps. This is because one cannot easily, and may not desire to, merge two running app instances, one that could be running natively and one that is being migrated. Thus, until the migrated app is brought back to its home device, an icon for the migrated app will exist on the guest device's launcher screen allowing for the user to resume the migrated app even after its been stopped.

Cross-Device App State Consistency. Once an app is migrated, it is guaranteed to have the latest and consistent snapshot of app state. When the user is finished with the app on the guest device, he may initiate a migration of the app back to its home device, thus resolving the inconsistency of app state between the two devices. If the user attempts to start the migrated app on the home device without having migrated it back, he is prompted with a message asking if he would like the app state from the guest device to be synced back to the home device or proceed while losing modified state on the guest device. Until an app has been migrated back to its home device, any security credentials allowing it to access online accounts will persist on the guest device until expiration or manual revocation.

Supporting Different Android Versions. Flux is capable of migrating apps between different kernel versions and minor Android version differences. Support for migration across major versions of Android would need to address two key challenges. The first is that an app using features only found in a newer Android API will be unable to migrate to an older version lacking those

features. It would be difficult to surmount this obstacle and doing so would likely place a dependency on the source device, e.g., require that the target device continue to use some of the source's services over the network. The second is that the private APIs of services used internally by the framework must maintain backward compatibility with previous versions. Currently, these APIs are commonly changed by Google in between versions.

Limitations. Apps that request their OpenGL context persist while in the background are unsupported by Flux. Apps are able to do this in Android by calling GLSurfaceView's `setPreserveEGLContextOnPause` method. Doing so allows them to cache textures, shaders, etc. in graphics memory so there is no display delay once the app moves back into the foreground. The downside of this is that the app consumes resources even while not visible and as such the feature is not commonly used. Unfortunately, if the context never goes away and apps expect it to remain, they may not use conditional reinitialization relied upon by Flux. Completely unloading and reloading graphics state becomes problematic in this case.

Apps that request to be run in multiple processes are currently unsupported by Flux. Because multi-process apps are relatively rare, this feature was simply not yet implemented. It can be added with modest additional engineering effort, as CRIU already supports checkpointing an entire process tree.

Migrating an app while it is interacting with a ContentProvider is currently unsupported, e.g., when an app is receiving data after querying the system for contacts information. In Android, data intended for use by multiple apps, such as contacts, can be shared using ContentProviders. ContentProviders expose an API similar to databases, with methods such as query, insert, and delete. This API is accessible via Binder and ContentProviders are essentially Binder services with short-lived app connections. As such, it should be possible to leverage Flux's Selective Record/Adaptive Replay for support, but due to the limited time frame during which an app is typically interacting with ContentProviders and the likelihood of it interfering with migration, we have not yet implemented or exhaustively explored support for this.

Only app-specific SD card data directories are migrated along with an app. Due to this, apps

accessing common SD card data at the time of migration will fail to migrate. Due to the potential size and quantity of files on the SD card, transferring them all is undesirable. Automatically transferring any open SD card files along with the app would allow these apps to migrate successfully, but any other common SD card files they were expecting to access would no longer be available. A potential solution could be to migrate the app and mount the home device's common SD card data as a network file system prior to restoring it, but this may not give the user the desired, or expected behavior.

Applying Flux to other mobile platforms. Although Flux is tailored to Android, the general design is applicable beyond it. Flux relies on three key platform characteristics: devices are utilized through system services and interacted with through a single IPC mechanism, app graphical resources can be released while the app is in the background, and the availability of an extensible checkpoint/restore mechanism. The first is a common mobile OS design paradigm. The third can always be overcome through engineering effort, and should be available for most Linux-based mobile OSes through CRIU. The second is perhaps the most problematic as any OS that does not already operate in this manner cannot easily be changed without breaking existing apps. For example, although iOS disallows apps from making OpenGL calls while in the background, apps are allowed to, and commonly do, retain their GL context. Removing their context while in the background would likely break most iOS apps. Existing work on checkpointing and restoring OpenGL state could be leveraged and improved upon to work around this requirement [34].

2.4 Evaluation

We have implemented a Flux prototype in Android and demonstrated its complete functionality in migrating unmodified Android apps across different Android devices, including the LG Electronics produced Google Nexus 4 phone and different hardware versions of the ASUS produced Google Nexus 7 tablet. The prototype has been tested to work with multiple versions of Android, including KitKat, the most recent version at the time of our evaluation. In migrating apps across devices with different screen sizes, Flux seamlessly migrates Android apps from home to guest

NAME	WORKLOAD
Bible	View page of the Bible
Bubble Witch Saga	Play witch-themed puzzle game
Candy Crush Saga	Play candy-themed puzzle game
eBay	View online auction
Flappy Bird	Play obstacle game
Surpax Flashlight	Use LED flashlight
GroupOn	View discount offer
Instagram	Browse a friend's photos
Netflix	Browse available movies
Pinterest	Explore "pinned" items of interest
Snapchat	Take photo and compose text
Skype	View contact status
Twitter	View a user's Tweets
Vine	Browse a user's video feed
Subway Surfers	Play fast-paced obstacle game
Facebook	Post comment on news feed
WhatsApp	Send text to friend
ZEDGE	Browse ringtones and select one

Table 2.3: Top free Android apps and how they were used prior to migrating.

device, including refreshing the app display to match the resolution of the target device.

We quantitatively measured the performance of our unoptimized prototype migrating and running a wide range of popular Android apps from Google Play. Our measurements were obtained using a Nexus 4 phone (Qualcomm Snapdragon S4 Pro APQ8064, Adreno 320 GPU, 2 GB RAM, 768x1280 pixel IPS LCD), a Nexus 7 (2012) tablet (NVIDIA Tegra 3 T30L, ULP GeForce GPU, 1 GB RAM, 1280x800 pixel IPS LCD), and two Nexus 7 (2013) tablets (Qualcomm Snapdragon S4 Pro APQ8064, Adreno 320 GPU, 2 GB RAM, 1920x1200 pixel IPS LCD). The Flux implementation used for our measurements was based on the Android Open Source Project (AOSP) version 4.4.2, the most recent version available at the time our measurements were taken.

To measure the cost of migration, we installed and ran eighteen different apps from the listing of top free Android apps from Google Play, including Candy Crush Saga, the long-standing most popular free game on Android. Table 2.3 lists the apps we used, along with a brief description of the workload used for each app. To demonstrate the ability of Flux to migrate across heterogeneous Android devices, we migrated these apps across all four Android devices in four different

combinations: (1) Nexus 7 (2013) tablet to Nexus 7 (2013) tablet to show migration using the same type of device on both sides, (2) Nexus 4 phone to Nexus 7 (2013) tablet to show migration from a smaller screen phone to a larger screen tablet, (3) Nexus 7 tablet to Nexus 7 (2013) tablet to show migration across two devices with very different hardware (GPUs, etc.) and kernel versions (3.1 and 3.4, respectively), and (4) Nexus 7 tablet to Nexus 4 phone to show migration from a larger screen tablet to a smaller screen phone, again with very different hardware and kernel versions. All devices were connected to a campus WiFi network. Before performing any migrations, all four devices were paired with one another.

Before and after each migration, a user used each app on the respective device based on the respective app workload. All but two of the apps, Facebook and Subway Surfers, were migrated successfully across all four different device combinations, with the visual layout of each app adapted to the screen size of the respective device after migration. Facebook could not be migrated because it is one of the few apps that is multi-process, and the Flux prototype currently does not support multi-process apps. Subway Surfer could not be migrated because it requests that its EGL context persist, a limitation discussed in Section 2.3.3. We provide detailed measurements for migrating the other sixteen apps to quantify the cost of migration.

Figure 2.12 shows the time required to migrate each of the apps across all four device combinations. Figure 2.13 shows the percentage breakdown of average migration times across the four device combinations. We breakdown the migration time into five stages: (1) preparation involves putting the app in the background to eliminate app-specific device state, (2) checkpoint involves checkpointing the app and its recorded log, (3) transfer involves verifying and syncing necessary file system state and sending the checkpoint image from one device to the other, (4) restore involves restoring the app from the checkpoint image, and (5) reintegration involves replaying calls to system services and bringing the app back to the foreground. The relative cost of each migration stage is fairly constant, with data transfer time dominating the cost of migration. As shown, over half the time on average is spent on the data and image transfer over WiFi.

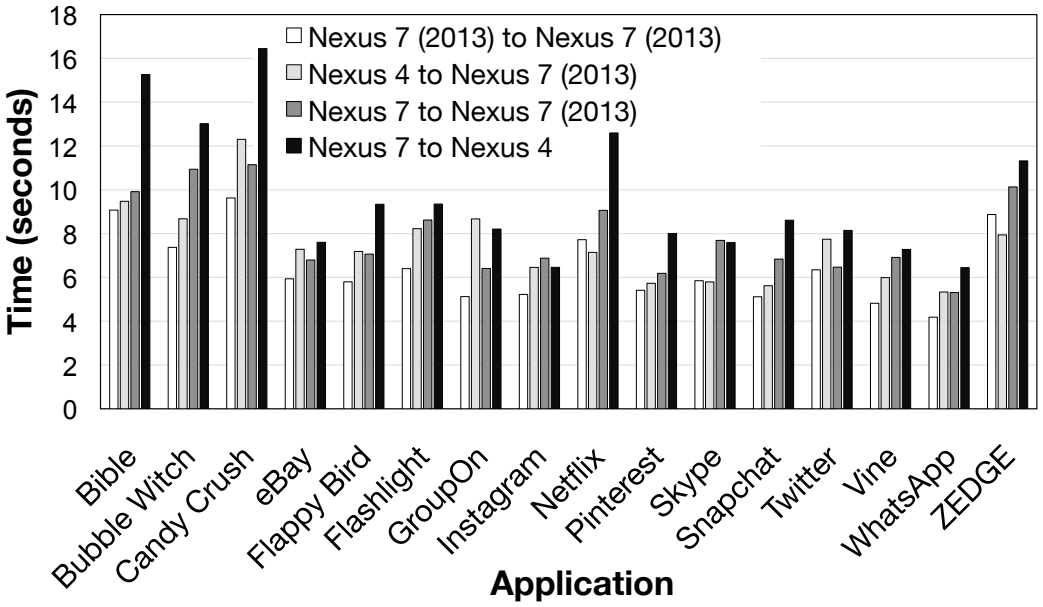

Figure 2.12: Overall migration times.

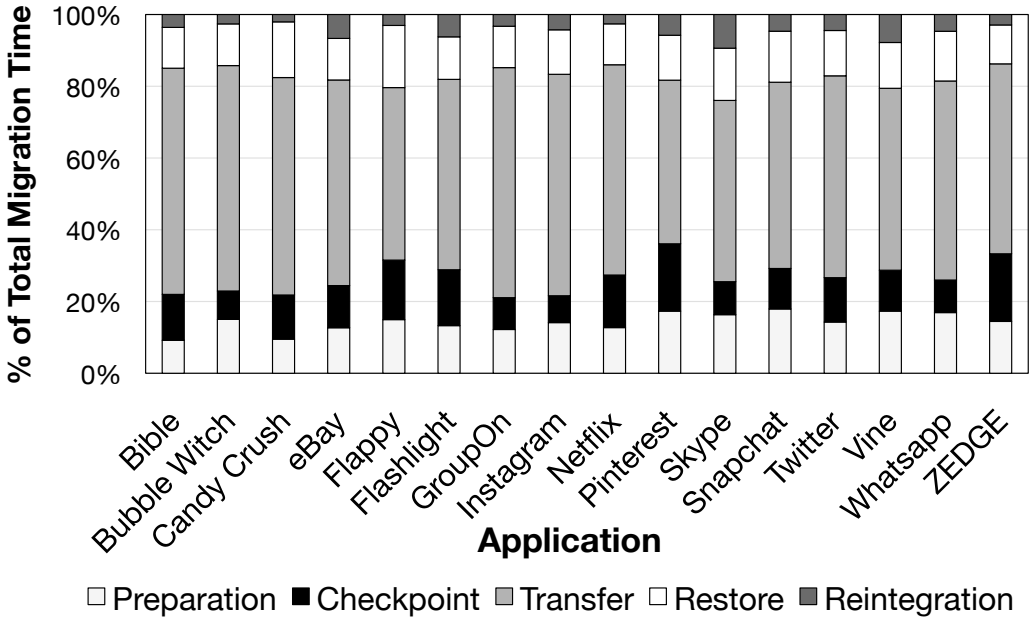

Figure 2.13: Breakdown of time spent during migration.

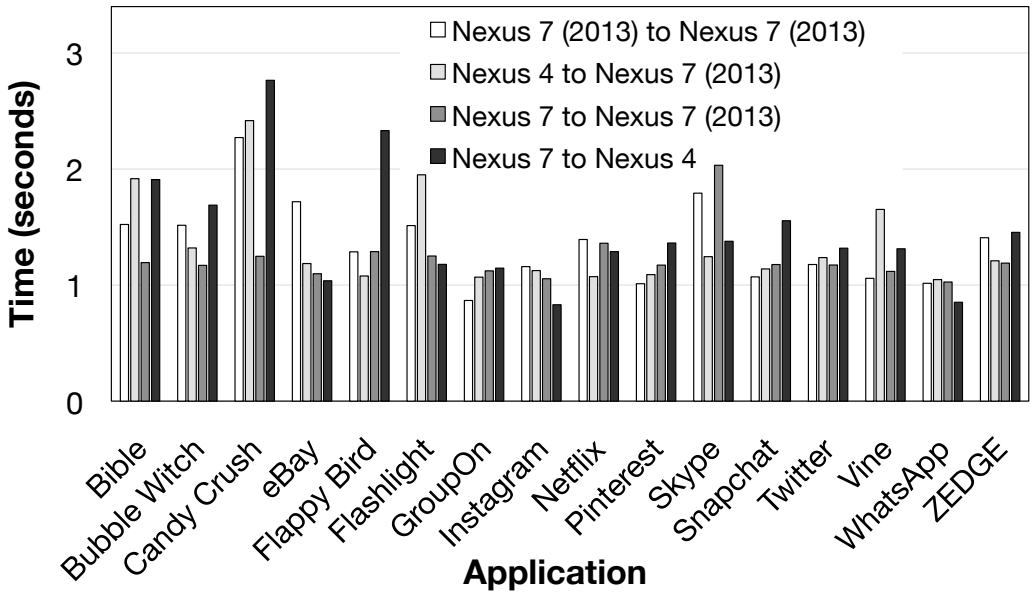

Figure 2.14: User-perceived migration time excluding data transfer phase.

Across all shown devices and apps, migrations required 7.88 seconds to complete on average. This is inclusive of the time required for data transfer in a congested, urban environment, as well as the preparation and checkpoint stages. However, the preparation and checkpoint stages will largely go unnoticed as they occur while the user is presented with the migration target menu and they make their choice. This results in a user-perceived average migration time closer to 5.8 seconds. Given that the data transfer stage is bound by the network bandwidth, it will continually improve as devices and wireless technologies evolve. For example, the latest mobile devices, such as the Google Nexus 5, feature 802.11ac wireless adapters. On 802.11ac capable networks, these devices can significantly outperform the 802.11n performance of the evaluated devices, especially the Nexus 7, which is only capable of operating on the extremely congested 2.4Ghz band. In the future, the data transfer stage could also be greatly reduced by deferring memory transfer using techniques such as post copy supplemented with adaptive pre-paging [35]. This also allows for the data transfer cost to be partially overlapped with the restore and reintegration stages. Looking ahead, to get a better idea of the potential migration times, Figure 2.14 shows the user-perceived time required for migration excluding data transfer, an average of 1.35 seconds. Note that our

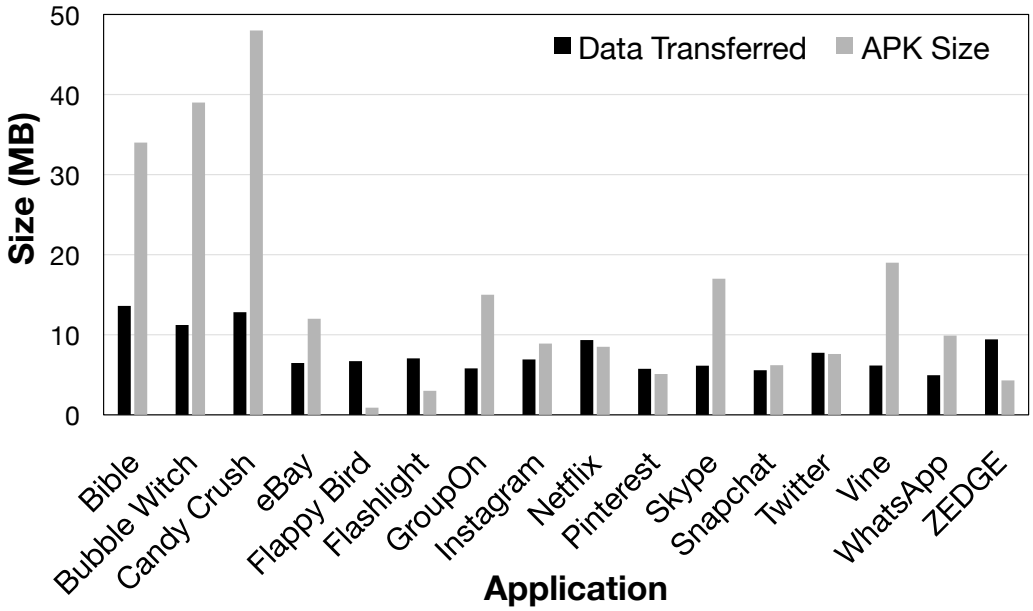

Figure 2.15: Amount of data transferred during migration.

prototype is not fully optimized and various migration stages can be improved. For example, Flux currently implements an unoptimized preparation for checkpoint that depends on the Android task idler to stop the app after we have placed it into the background.

Figure 2.15 shows the average data transferred to migrate each of the apps between devices. We also show the APK size of each app for reference. The amount of data transferred is dominated by the size of the checkpoint image, and in our tests, the compressed data directories sync and record log never exceeded a combined 200KB. None of the migrations required transferring more than 14MB of state during the data transfer stage. Comparing Figure 2.15 with Figure 2.12, the migration times are generally correlated with the data transfer sizes. We can loosely say that the larger the app's install size is, the longer it can be expected to take to migrate.

To demonstrate that the recording costs of Flux are modest, we ran the Quadrant Standard [36] and SunSpider [37] benchmarks on both Flux and vanilla Android. Figure 2.16 shows the results of running the benchmarks on all three types of devices normalized to AOSP and indicates that the overhead is negligible in all cases.

To get a real-world idea of the challenges Flux faces, both in migration performance and sup-

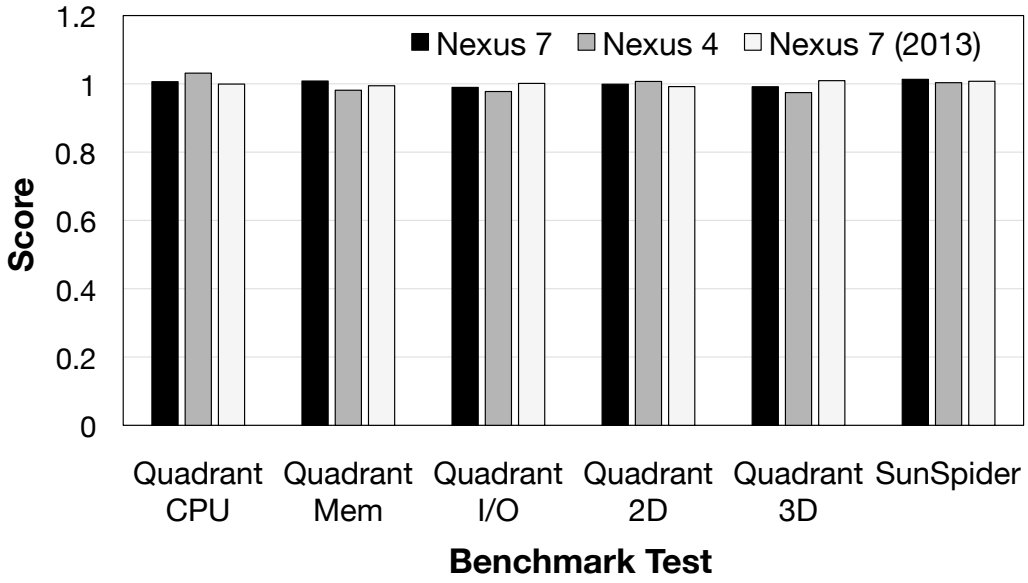

Figure 2.16: Quadrant Standard and SunSpider benchmark results normalized to AOSP.

port of apps, we analyzed several hundred thousand free Android apps in Google Play. We leveraged PlayDrone [38] to crawl the Google Play store, download the metadata and APKs for a collection of 488,259 apps, and decompiled the APKs to analyze their sources. Since Flux cannot migrate apps which choose to always retain their graphical context, we parsed the sources to identify those that explicitly call Android's `setPreserveEGLContextOnPause`. Of the roughly half million apps we downloaded, this call is only made by 3,300 of them. This indicates that only a small percentage of the apps in Google Play use this feature, and that the Flux approach is expected to work for the vast majority of apps.

Since the cost of pairing devices before migration involves transferring APKs, we also analyzed the collection of apps in Google Play to measure their installation sizes, information included in the metadata associated with each app. To verify that the installation size is a good measure of the actual size of the app APKs, we looked at a random selection of APKs from Google Play and compared their actual size to the installation size. The installation size and actual APK size matched in all cases. Figure 2.17 shows the cumulative distribution function of all the apps versus their installation size. Roughly 60% of the apps are less than 1 MB in size, and roughly 90% of

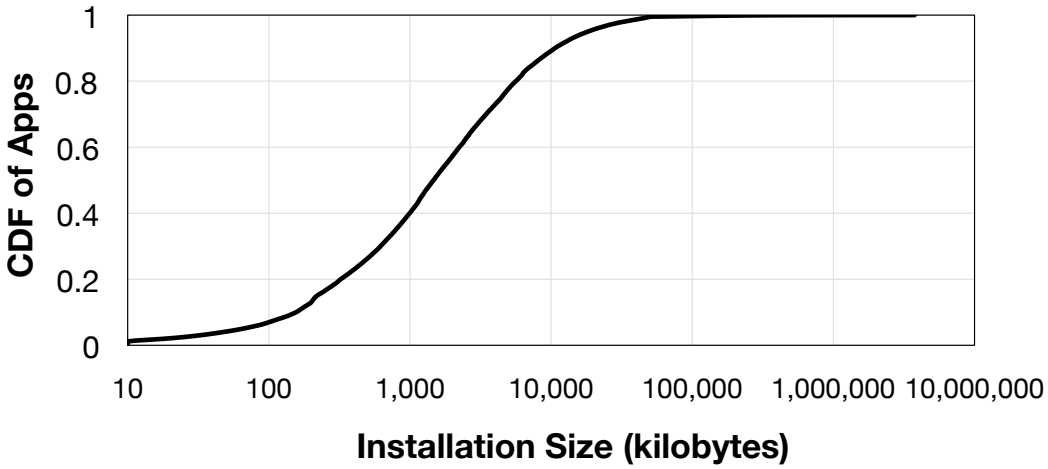

Figure 2.17: Installation size of Google Play apps.

the apps are less than 10 MB in size.

We also measured the pairing costs for the various devices we used for migration. Pairing consists of a constant data cost component and a cost that scales linearly with the number of installed apps and their install size. The constant data is comprised of a device's system libraries, frameworks and apps. When pairing a Nexus 7 to a Nexus 7 (2013), both running KitKat, the total constant data size that must be synced was 215MB. After accounting for identical files on the target device that can be hard-linked, this is reduced to 123MB. The compressed delta that must be transferred is 56MB.

2.5 Related Work

Application migration has been extensively studied across a broad range of desktop and server computing systems. Many research operating systems (OSes) have implemented support for process migration, including Accent [39], Amoeba [40], Chorus [41], DEMOS/MP [42], MOSIX [43], Sprite [44], and V [45]. These OSes provide a global namespace and location transparent execution allowing processes to migrate freely across machines. Migrated processes often rely on their home machine for IPC, open files, and system calls, forever tethering them to another machine. None of these approaches are designed for mobile devices, and do not address the key device heterogeneity

problems to support migration on mobile devices.

Arguably the most popular migration approach today is VM migration, leveraging virtual machine monitors (VMMs) to virtualize at the hardware level and encapsulate an entire OS [46, 47]. These approaches are used in server and cloud environments, where whole OS virtualization and migration is practical and works well. However, using VMs on mobile devices has been problematic, as existing approaches [48] provide no effective mechanism to enable apps running in VMs to directly leverage hardware device features without substantial performance degradation, especially for apps using 3D accelerated graphics [11, 49]. As a result, no VM-based solutions exist for enabling app migration across commodity smartphones and tablets.

There has been significant research in checkpoint-restore approaches which have been used for migration, spanning the application-level [50, 51, 52], library-level [53, 54], library OS-level [55, 56] and kernel-level [57, 58, 7, 59, 60, 33, 5]. None of these approaches work for commodity smartphones and tablets, and do not address the key device heterogeneity problems to support migration on mobile devices. Application-level mechanisms [61, 62], while efficient, are non-transparent, require application-level modifications, and may require nonstandard programming languages [63].

Library checkpoint-restart mechanisms require that applications be compiled or relinked against special libraries. Unlike Flux, such approaches do not capture important parts of the system state, such as interprocess communication and process dependencies through the OS, and do not support significant changes in underlying hardware or the plethora of devices found in mobile platforms.

Library OS approaches encapsulate an entire OS at the user-level to make checkpoint-restore of OS and application state easier across desktop computers, but rely on remote display mechanisms [55], limiting graphics performance. It is unclear how these systems might support app migration across mobile devices. Like distributed OSes, the hard part is migrating across heterogeneous graphics hardware; any library OS attempting to migrate from a desktop to a tablet would need to adopt exactly the kind of mechanisms that are provided by the Flux solution.

Kernel-level approaches include those that require entirely new OSes [57, 58], limiting their

deployment, and those that work with commodity OSes such as Linux [7, 33, 5], which led to the current CRIU checkpoint-restore support in Linux [30]. Flux builds on CRIU but specifically targets mobile devices, focusing on providing the necessary hooks to extract and reintegrate application state from Android-specific software device drivers, as well as interactions with system services, and hardware devices to support migration across disparate devices.

Recently, an Android-specific checkpoint-restore project was created for restoring the Android Zygote process for faster booting [64]. However, the project does not support checkpoint-restore of interactive or GUI-based processes and therefore does not support Android apps, supports only same-device checkpoint and restoration, and does not support interaction with hardware devices.

There has also been significant research in record-replay approaches [65, 66, 67, 68, 69, 70, 71], in some cases to even replicate application state across different computers [72]. Unlike Flux, these systems assume a homogeneous environment and are not designed to allow replay with any modifications to the recorded execution.

Other approaches enable replay with varying degrees of modifications from the recorded execution. Crosscut [73] can reduce the information recorded in a log so that, for example, sensitive information can be purged before replay. Scribe [74] replays a recorded application execution until a specified point, and then transitions to live execution instead of replaying the rest of the log. Racepro [75] detects process races due to dependencies in the ordering of system calls by recording an application execution to a log, identifying a pair of system calls that may be racy, truncating the log at the occurrence of the pair of system calls, inverting their order, and replaying the truncated log with the reordered system calls. A few record-replay systems allow new code to be run while replaying a recorded execution [76, 66]. However, this new code cannot have any side effects on the program. More recently, Dora [77] allows transparent mutable replay of application execution even when applications change. Recent work also applies record-replay to graphical contexts by leveraging a record-prune-replay mechanism capable of restoring an OpenGL state by replaying the minimal number of calls necessary [34]. Flux differs from previous approaches in that it targets mobile service invocations and leverages their semantics to guarantee correctness as device state

changes, adapts to changes in hardware, and is much lighter weight, making it more suitable for mobile devices.

2.6 Summary

Moving computation across glass surfaces has been the vision of science fiction for decades. Recent advances in mobile device computing capacity as well as the proliferation of glass surfaces in cell phones, phablets, tablets, smart TVs, and smart watches, all running similar OSs, will likely enable new forms of computing interactions that extend beyond a single device. We have demonstrated that such experiences are indeed possible with Flux. A user can move apps—mid execution—across Android-based mobile and tablet devices. Our design focused on minimizing the intrusiveness on existing mobile OS stacks and apps. At same time, we wanted to leverage the clean separation between apps and services within a mobile OS to allow for a fluid migration experience where apps can gracefully adapt to changes in hardware devices. We have showed that many popular apps can be migrated without any modifications. In the process, we have fully captured the various overheads in migrating an app.

Chapter 3: AnDrone: Virtual Drone Computing in the Cloud

3.1 Introduction

Recent advancements in drone technology have allowed the use of drones, unmanned aerial vehicles (UAVs), in applications from aerial photography to package delivery, as well as a wide array of surveying, inspection, and security applications. Smaller, feature limited consumer drones have become more affordable and user-friendly, but still maintain a steep learning curve, requiring significant time investment before becoming proficient in their use and remain prohibitively expensive for infrequent use. Larger and more capable drones remain expensive and out of reach for most consumers, both in terms of cost and complexity. For all drones, users must learn and follow regulations such as where drones can and cannot fly, registration, and licensing, increasing the burden placed on users wanting to use a drone, particularly those with only the occasional use for them. As drone usage continues to rise [78], it is likely that these burdens will only further increase as the need to manage limited airspace only grows. These operational costs coupled with the limited flying time available with most drones due to energy constraints make the time that a drone is actually flying quite valuable. Each flight should be leveraged to its fullest to offset these costs, but despite this, drones today are typically monotasking with a single task assigned for each flight. Given two tasks, there will be two separately owned drones used, each requiring proficient operators and airspace to operate, even if the two tasks would involve the same flight path.

To address these challenges and make drones more widely available, we introduce AnDrone. AnDrone is a drone-as-a-service solution that makes drones accessible in the cloud to interested third parties. With companies like Amazon, UPS, and DHL investigating mass rollouts of delivery drones [79, 80, 81], AnDrone can enable these drones to also be made available to interested third-parties via the cloud to provide additional services. A drone previously tasked with a simple

delivery can now simultaneously survey vehicle traffic conditions for a local news company while en route to a delivery, routinely survey a construction site's progress, or photograph a property for a real estate agent. AnDrone enables these use cases without requiring the user to obtain additional hardware or have in-depth knowledge about drones.

AnDrone provides third-party users with lightweight virtual drones that can be configured in the cloud with various apps and services of interest to a user, then safely deployed and multiplexed on real drone hardware. With AnDrone, multiple third-party virtual drones may run simultaneously and continuously throughout a single physical flight as the drone travels from one waypoint to another. At each waypoint, the virtual drone can be given control of the drone and additional device access can be granted allowing the virtual drone to complete any required tasks. AnDrone takes advantage of the observation that computational costs on drones are cheap compared to the operational and energy costs of putting drones in the air, making it is very efficient to multiplex multiple virtual drones on a physical drone to maximize the utility of drone flight time.

AnDrone introduces a novel Linux container architecture to support and isolate different drone execution environments. Unlike, traditional hardware virtualization approaches [4, 3, 2, 82], AnDrone's lightweight container approach [83, 5] pairs well with drone hardware that tends to lack hardware virtualization support and be resource constrained given size, weight, power, and cost considerations. Each virtual drone has its own containerized Android Things environment with which to run its tasks and can provide online interactive access to the drone during flight. AnDrone utilizes Android Things [84] to offer users a familiar and well-known environment with many existing apps, and developers the ability to leverage a large existing base of code, libraries, development tools, and resources, all tailored for Internet of Things (IoT) systems such as drones. To control physical device access to isolate virtual drones from one another and preserve drone hardware safety, AnDrone introduces a device container for managing and multiplexing device access and a real-time Linux flight container for drone flight control.

The device container isolates devices from virtual drones by encapsulating all physical drone devices in a separate isolated execution environment. Only the device container has access to

physical devices, allowing it to be used to gate and multiplex access to those devices from virtual drones. Isolating devices in their own execution environment is made possible by leveraging how apps in Android Things interact with devices via higher-level system services [85]. These services allow apps to be transparently decoupled from low-level device implementations and interfaces so that devices can be separated from the rest of the Android Things execution environment. Unlike other container-based hardware multiplexing approaches [11], our approach requires no explicit per-device support, significantly reducing the effort needed to support new platforms and devices. The device container further creates the illusion for the physical devices that each such device is only being used by one task at a time, providing easy compatibility with existing drone-specific software and hardware stacks which are often not designed to support multiplexing. By allowing virtual drones to remain independent of physical devices, they can also be easily moved as needed to different physical hardware.

The flight container mirrors the device container approach and isolates the critical real-time flight software stack from virtual drones by encapsulating all flight control logic in a separate isolated execution environment. Only the flight container has access to the physical hardware for flight control, allowing it to be used to gate and multiplex access to flight control from virtual drones. The flight container also allows a different execution environment for the flight software stack, which is crucial for software compatibility as it is based on real-time Linux, not Android. A simple network proxy-based approach enables Android Things virtual drones to interoperate with the real-time Linux flight container.

AnDrone leverages its device and flight containers to provide location-based and conditional drone control. Access to devices such as cameras, camera gimbals, sensors, and GPS can be conditionally granted to virtual drones. Similarly, virtual drones can be geofenced and restricted to operating within a defined set of control parameters. This is used to provide device isolation among virtual drones; a virtual drone restricted to operate in one locale can be prevented from operating in another. This is also used to provide operational safety, for example disallowing overly aggressive maneuvers and enforcing obstacle avoidance. Drone providers can customize

the degree of control a user is given over a drone, even restricting it to only operating in a guided mode wherein the drone can only be given destination coordinates and a velocity with which to reach it. With various device restrictions possible, multiple third parties may securely run tasks throughout a single flight and operate a drone in-turn, without interference with each other or the flight stack, fully maximizing the potential of a drone's flight.

We have implemented an AnDrone prototype supporting multiple Android Things virtual drones on drone hardware based on the Raspberry Pi 3 Model B [86] and Emlid Navio2 [87] daughterboard. Our experimental results demonstrate runtime performance overhead of less than 1.5% for a single virtual drone, a negligible effect on drone energy usage, the ability to multiplex multiple virtual drones while ensuring low-latency performance within 300µs of an idle system and sufficient to meet the real-time requirements of drone flight, and that untrusted third-party software may run in virtual drones without undue risk to the physical drone.

3.2 Usage Model

With AnDrone, users with little to no drone experience can obtain a virtual drone equipped with premade apps to control the drone to accomplish a desired task. For basic drone service, users interface with the AnDrone web portal to order and configure a virtual drone, AnDrone assigns the virtual drone to a physical drone to perform the desired task, then the data from the drone is uploaded back to the AnDrone web portal for the user to access. More advanced options are also available, for example to provide interactive control of the drone during flight.

To order a virtual drone, a user accesses the AnDrone web portal, shown in Figure 3.1, selects one or more way waypoints, locations where the drone should go, and specifies a desired date and time range for using the drone. A list of possible drone types available is then presented for selection, e.g. drones specializing in obtaining video, drones equipped with specialized sensors, etc. Apps can then be uploaded on to the virtual drone, including by selecting from existing apps available in the AnDrone app store. For example a real estate agent who wants aerial photography of a house can go to the AnDrone app store and find an app that will do this. It could be a basic

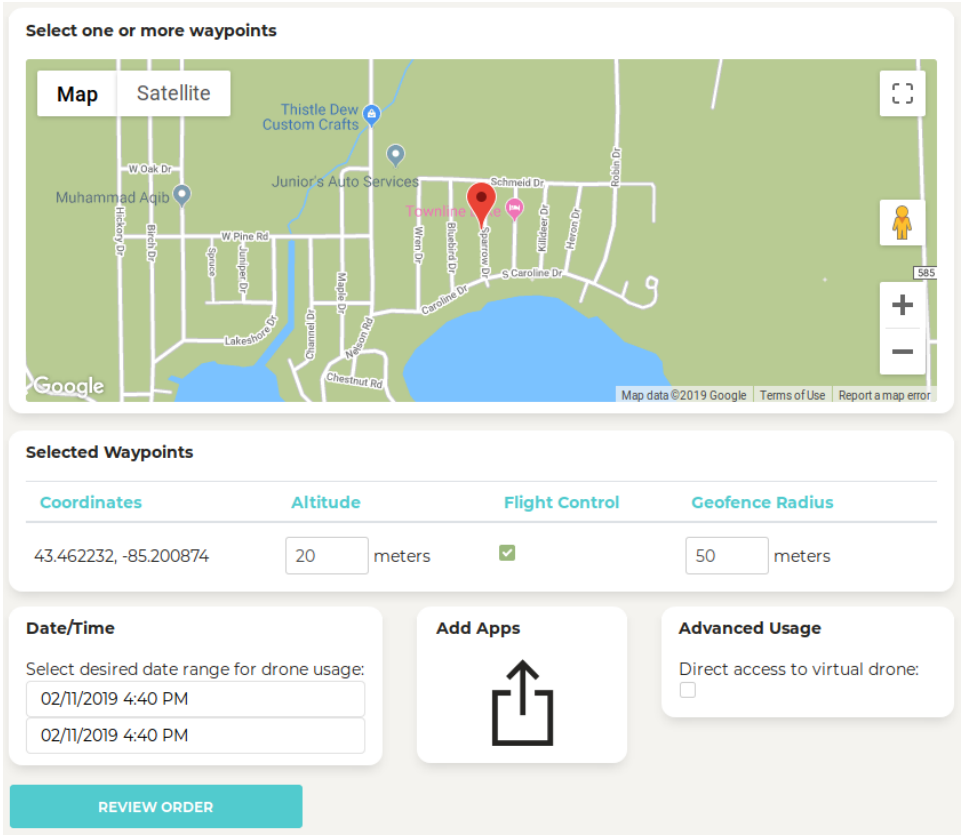

Figure 3.1: Snippet of the AnDrone web portal interface.

app that simply circles a geographic location, a more advanced app leveraging computer vision to obtain better results, or even a service-based app offering another company's pilot to manually obtain results. AnDrone leverages Android Things to make it easy for both app developers and users to build on a familiar and established app ecosystem.

Once an app has been selected, the user will use the portal to supply the app with any arguments it requires, e.g., an area on a map to survey. Any further interaction with the drone or app after take off is app-specific. The app may supply a front-end that the user can run on their smartphone or in a web browser to see additional status information or make additional input, or it may act fully autonomously and simply offer the user files it generates. The virtual drone will return flight control once it has completed its task. The user will be informed once the drone has finished its flight, and emailed a link to any files the app generated for them.

If a user requires more advanced functionality, direct access to the virtual drone can be provided

instead of just specifying an app to run. When ordering a virtual drone for advanced usage, the user also specifies any devices they need access to and whether they need access to those devices both at and between waypoints or just while operating at a waypoint. If immediate usage is requested of a virtual drone, AnDrone will provide the user with an estimated operating window of when to expect the drone to arrive at the first waypoint so the user can then take over control of the drone. If the user is flexible with regard to when the drone launches, AnDrone will provide an estimated operating window a day in advance of the flight to confirm if it is acceptable to the user. Once the drone takes off, the user is notified via email or text message and the portal provides provides access information for the virtual drone, notably its IP address and port information and how the user may connect to it, much like any recently deployed cloud-based server. The user can then access the virtual drone remotely and run tasks on the virtual drone throughout the entirety of its flight, but flight control is only provided to the user at the specified waypoints of the virtual drone. If flight control is not requested at a waypoint, AnDrone will simply fly the drone on to the next waypoint after arrival. Such waypoints are useful to, e.g. guide a virtual drone along a highway to survey traffic. If flight control is given, the user may return control of the drone at any time via the portal or an app running in their virtual drone. Like other Android systems, device access is provided by Android drone apps. For example, an app running on the drone can forward the camera feed to a client app running on the user's smartphone. All communication between the drone and the user takes place via a cellular internet connection.

It would be unsafe to not enforce restrictions on a virtual drone's control of a physical drone. So once AnDrone hands over control to the virtual drone, the drone is geofenced and restricted to operating within a defined set of control parameters. The extent of these restrictions is flexible and can include multiple aspects of drone flight. For example overly aggressive maneuvers can be disallowed, forced obstacle avoidance can be added, flight modes can be restricted, etc. This allows for a range of functionality varying from allowing users full control of the drone with only basic restrictions on extreme maneuvers, to only allowing the drone to operate in a mode such that it is given destination coordinates and a velocity with which to reach it. With such restrictions the

drone can still be pathed wherever the user wants, but always in a predictable manner. The size of the geofence that is applied to the drone is requested by the user up to a maximum size when ordering the drone via the AnDrone portal, with a default size provided.

In addition, AnDrone's containerized design in combination with its device access control ensures privacy and isolation among virtual drones. For example, it is possible that a user A's virtual drone has multiple waypoints and requests access to devices such as the camera while the drone is operating between them. While routing between these waypoints, another user B's virtual drone waypoint may be visited. In such cases, for privacy and conflicting device control reasons, user A's device access will be suspended by default until the drone has finished at user B's waypoint. AnDrone assumes that a user would generally not want another party to have access to the drone's camera or microphone while operating at the user's waypoint.

Like all cloud resources, AnDrone billing is based on usage, but unlike, e.g. a cloud server where time can be used as the billing unit, a drone's flight time is limited and can vary greatly with both the type of drone and how the drone is operated. AnDrone can bill traditional cloud services such as storage or network bandwidth based on regular usage, but bills drone usage based on energy consumption, like a traditional energy utility service. Energy is used for billing drone usage because of its direct correlation with the most critical resource for drones, as well as the ability to leverage the familiarity of energy utility pricing. Estimates of flight time based on energy usage [88, 89, 90, 91, 92] are provided to the users when ordering a drone and the user's specify a maximum billing charge, which in turn specifies the maximum energy the user's drone can consume at their waypoints.

It is possible that the task a user wishes to perform (either via an app or direct access) is unable to be completed on a drone for various reasons, including exceeding the user's maximum billing charge or unpredictable events such as inclement weather. In these cases, virtual drones are instructed to save their current state so that they can be resumed on a later flight.

3.3 Virtual Drone Definition

To be able to place a virtual drone on a physical flight the following must be known about it: where it is to operate, how much energy it may use, how long it can operate, which devices are needed, when those devices are needed, and what apps should be installed and run. To accomplish this, AnDrone defines a virtual drone as a JSON specification in combination with an Android Things container image. Upon receipt of a new virtual drone JSON specification, AnDrone creates a clean Android Things container and installs any specified apps in it. From then on, the JSON specification and container defines the entirety of the virtual drone. A virtual drone definition is fully self-contained and can be easily reinstated on any drone or even non-drone hardware so long as the CPU architecture matches and the kernel is equipped with Android's kernel features. Each virtual drone container image consists only of its differences from a base virtual drone image, allowing for minimal storage requirements when running multiple virtual drones and storing them offline.

Figure 3.2 is an example of an AnDrone virtual drone JSON specification, which has seven components. First, the specification has a list of *waypoints* a virtual drone is to visit, each of which is defined by a desired *latitude*, *longitude*, *altitude*, and *max-radius* in meters, which defines a spherical volume from the given waypoint coordinates. Together these parameters define a geofence that will be applied to the virtual drone's control of the real drone, if flight control has been requested. *Max-duration* in seconds and *energy-allotted* in joules combine to specify the maximum time and energy allotted for the virtual drone to operate at all of its waypoints, whichever is exhausted first dictating when control must be taken away. Max-duration is specified in such cases where virtual drones are allowed to land, thus preventing them from idling on the ground indefinitely. *Continuous-devices* specify the list of devices that the virtual drone should have access to continuously once its first waypoint is reached until it completes operation at its last waypoint. *Waypoint-devices* specify the devices that the virtual drone should have access to only while operating at waypoints. Waypoint-devices are prioritized above continuous-devices,

```
{
  "waypoints": [
    { "latitude": 43.6084298,
      "longitude": -85.8110359,
      "altitude": 15,
      "max-radius": 30
    },
    { "latitude": 43.6076409,
      "longitude": -85.8154457,
      "altitude": 15,
      "max-radius": 20
    }
  ],
  "max-duration": 600,
  "energy-allotted": 45000,
  "continuous-devices": [
  ],
  "waypoint-devices": [
      "camera",
      "flight-control"
  ]
  "apps": [ "com.example.survey.apk" ],
  "app-args": [
    { "com.example.survey": {
        "survey-areas": [
          { "43.6084298,-85.8110359": [
              [43.6087619, -85.8104110],
              [43.6087968, -85.8109877],
              [43.6084570, -85.8110225],
              [43.6084240, -85.8104646]
            ],
          { "43.6076409,-85.8154457": [
...
}
```

Figure 3.2: Virtual drone definition for example construction site surveys.

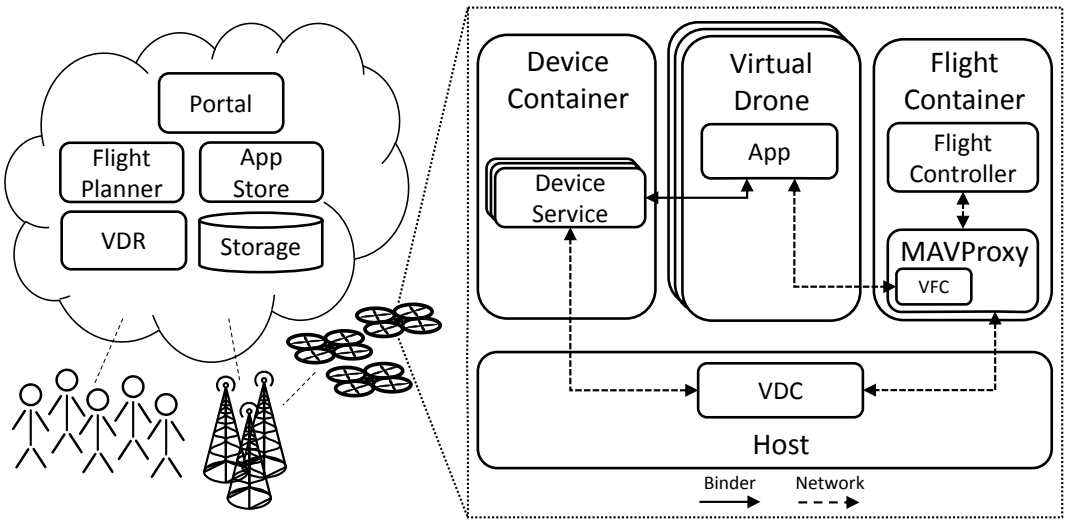

Figure 3.3: Overview of AnDrone architecture.

so continuous-device access is susceptible to temporary removal should another party's virtual drone's waypoint be visited in between specified waypoints. Flight control can only be specified as a waypoint device, not a continuous device. *Apps* specifies a list apps that should be installed in the virtual drone's container. *App-args* specify the arguments that should be passed to apps when they are started as given by the user when the drone was ordered via the AnDrone portal, in this case being sets of latitude and longitude pairs defining geographic regions that will be surveyed for each waypoint.

3.4 AnDrone Architecture

To support virtual drones, AnDrone pairs a cloud service for configuring, allocating, and storing virtual drones offline, with an onboard drone virtualization architecture to safely share physical drone hardware during flight while restricting access to the drone overall to secure it from untrusted third parties. Figure 3.3 shows the overall architecture of AnDrone. We provide a brief overview of the cloud service, then focus the rest of our discussion on the drone virtualization architecture components and how they interact with the cloud service.

As shown in Figure 3.3, the cloud service has five components: the AnDrone web portal users use to order their virtual drones, the AnDrone app store that provides apps for virtual drones,

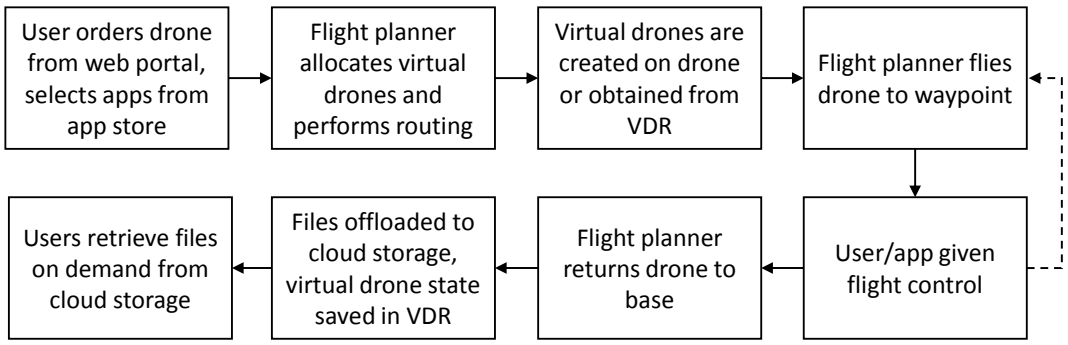

Figure 3.4: AnDrone workflow.

general storage for drone flight data, a virtual drone repository (VDR) which stores preconfigured virtual drone definitions for later use or reuse, and a flight planner which allocates virtual drones to physical drone flights and autonomously pilots drones from waypoint to waypoint. AnDrone's flight planner is based on the multirotor drone energy consumption model and the drone delivery routing algorithm developed by Dorling, et al. [92] for assigning deliveries to a fleet of drones to minimize delivery time subject to a drone fleet size constraint. AnDrone assigns virtual drones to physical drones using this model and algorithm by specifying the drone fleet size, using waypoints as delivery locations, and adjusting the energy cost to account for the energy allocated for virtual drones at their waypoints. A limitation of the algorithm is that it treats all waypoints independently, so users may not prescribe that waypoints be traversed in a specified order and the algorithm may decide to visit waypoints of one virtual drone in the middle of a set of waypoints of another virtual drone. Providing a planner algorithm that can support waypoint ordering and grouping is an area of future work. During flight, the cloud service communicates with drones over cellular internet as current LTE performance is already sufficient for cellular based drone control [93]; future cellular technology is being developed with mission critical drone usage in mind [94]. Figure 3.4 shows the workflow of an AnDrone flight and where each cloud service component is involved.

The onboard drone virtualization architecture is built on a Linux operating system (OS), given that drones are primarily ARM-based and Linux is the dominant OS for ARM devices. To enable running multiple virtual drones alongside a real-time Linux-based flight stack, AnDrone's virtualization architecture uses Linux containers to support running multiple variants of Linux at the same

time, including Android Things. AnDrone containerizes all Linux instances to provide isolation among them and manage their resources as necessary to ensure the reliability and performance of all containers. To provide real-time support for containers, AnDrone's Linux kernel is augmented with the PREEMPT_RT [95] patches to make it fully preemptible to minimize latencies for real-time tasks. Remote access to containers is provided by tunneling all communication over a per-container virtual private network (VPN), allowing potentially insecure protocols, such as those used by drone flight controllers, not originally intended for use across the Internet to now be used securely over cellular internet communication.

By relying on containers instead of traditional hardware virtualization [96, 97, 98, 99, 3], AnDrone removes the need to emulate numerous sensor devices, potentially introducing unacceptable latencies, and expands hardware compatibility to devices without hardware virtualization support. This is essential since due to the size, weight, power, and cost considerations drone hardware tends to be resource constrained and lacking the virtualization capabilities familiar to server hardware where virtual machines are commonly used. Additionally, AnDrone is able to maximize the limited resources of drone hardware by avoiding the need to run multiple full OS instances. By leveraging Android Things, an Android variant specifically designed for resource constrained IoT devices, AnDrone offers app developers a well-known environment and off-the-shelf reuse of Android apps and code inside a minimal, more resource-efficient OS than stock Android. Additionally, with out-of-the-box support for single board computers like the Raspberry Pi, Android Things offers better hardware support than stock Android for devices commonly used in drones.

As shown in Figure 3.3, the virtualization architecture has four main components: the virtual drone containers loaded onto the drone hardware, a device container for multiplexing device access, a flight container for virtualizing and multiplexing flight control, and a virtual drone controller (VDC) that manages virtual drones. We discuss each of these components in further detail below.

3.4.1 Virtual Drone Containers

Each virtual drone container appears to applications as an independent Android Things instance which is isolated from other virtual drone instances. For efficiency on the drone, virtual drone containers are managed using Docker [9] so that each container consists of common read-only base disk images layered together with a writable layer on top [100, 101, 102]. Common read-only base disk images can be shared across virtual drones, making virtual drones easier to manage and reducing storage costs. Docker also simplifies management by providing built in commands that enable AnDrone to easily move virtual drones back to the cloud and to other drone hardware as well as store them offline in the cloud. In addition, Docker enables AnDrone to prevent abuse and excessive consumption of resources, which can interfere with other virtual drones by allowing AnDrone to place restrictions on the resources each virtual drone can use.

While Docker is useful for supporting multiple Android Things virtual drones, it is not sufficient. Unlike traditional desktop and server computing environments, Android, and platforms that it runs on such as smartphones and IoT systems, incorporate a plethora of devices that applications expect to use. Some devices can be easily virtualized because they need not provide much of the original device functionality. For example, Android cannot be run without a graphical user interface and expects to be able to access a framebuffer device. Since drones are headless, the framebuffer contents are not actually displayed. In this case, each container can be simply given a virtual framebuffer device to use rather than the real one, and the virtual framebuffer device can just be a memory region in which contents can be written. No actual hardware device support is needed. However, for more complex devices that actually need to provide full featured functionality, existing approaches provide no effective mechanism to enable apps to directly leverage these device features from within virtualized environments, whether they be traditional virtual machines (VMs) or Docker containers. These devices are instead intended to be used directly by a single Android instance and cannot be used by multiple instances simultaneously, which AnDrone requires for supporting multiple simultaneously running virtual drones.

One key Android device is not actually a hardware device, but a software abstraction, namely

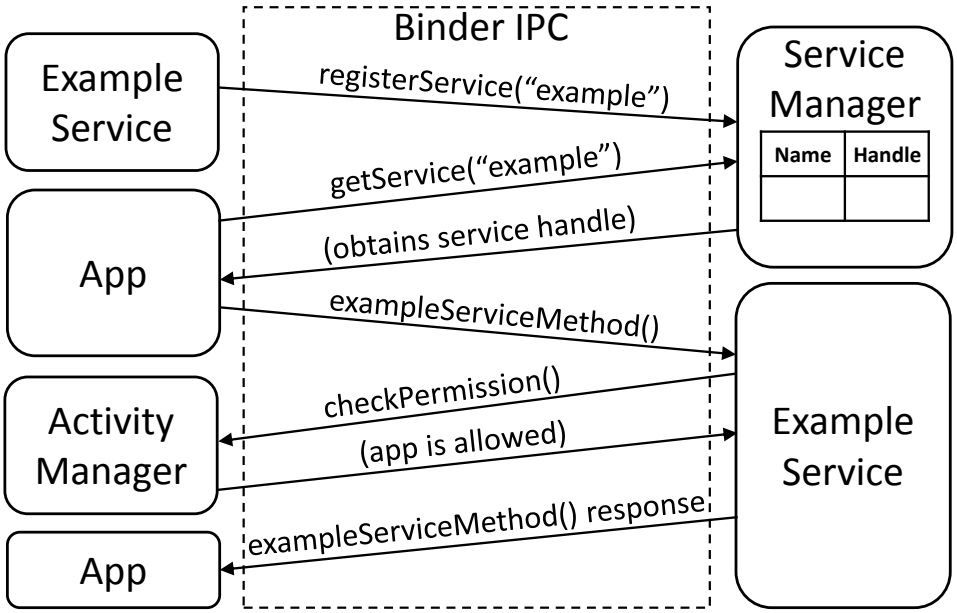

Figure 3.5: Android Binder service communication.

Android's Binder interprocess communication (IPC) device. Binder is a software abstraction that functions as Android's primary IPC mechanism and is utilized by processes via various ioctl system calls. Binder inherently provides isolation as no communication can occur between a client and a service without first obtaining a handle to that service; services exist as *nodes* that clients reference via an integer-based per-process *handle*. To obtain a handle to a node, a client must be given it by the node itself or someone who already has a handle to that node. In Android, services register themselves with the userspace *ServiceManager*, Binder's *Context Manager*, which itself is always obtainable through Binder via the handle 0. The ServiceManager retains a mapping of handles to corresponding names of services given at registration time. Apps can obtain handles to desired services by requesting a reference from the ServiceManager, as shown in Figure 3.5. Binder only allows one Context Manager, which offers handles to all services. However, AnDrone runs multiple virtual drone containers, each an Android Things instance, and each expecting to be have their own Context Manager offering their own services.

To achieve this, we add device namespaces [11, 49, 103] to Binder to isolate the Context Manager to a per-container level, allowing each virtual drone instance to have its own Context Manager. When a ServiceManager registers as a Context Manager, Binder identifies the container

from which the ServiceManager registers so that subsequent references to the container's handle 0 will reference the respective container's own ServiceManager instead of one global one. Since Binder does not allow access to services without access to their respective handles, the end result is that each container's clients and services are isolated from those in other containers.

While device namespaces are useful for enabling the Binder device to operate in the context of virtual drone containers, this is not as useful for actual hardware devices that need to deliver full functionality, in some cases involving complex and proprietary device drivers. Augmenting these complex implementations with device namespaces would be problematic both due to complexity and lack of availability of source code. While a virtual device could be introduced, it would still need to provide access to the actual hardware device functionality and could not simply be a dummy virtual device, which comes around to the original question of how to multiplex the hardware device among virtual instances.

3.4.2 Device Container

Unlike traditional desktop and server systems, Android IoT systems are highly vertically integrated in which several layers of software are involved on a given system to offer a tall interface from apps to hardware devices. Apps are written in Java and call Java frameworks, which function as libraries that provide the core public APIs used by developers for Android functionality including accessing devices. Frameworks use Java Native Interface (JNI) to package up calls and pass them through Android's Binder IPC mechanism to communicate with Android system services, which are system processes that run in the background and are used to manage devices. Apps do not interact with hardware devices directly, but instead via system services.

Ideally, these device services can be used to multiplex hardware for multiple containers as they are already designed to multiplex access to hardware devices from multiple processes. In a vanilla Android instance, system services would run as part of the Android instance. However, with multiple Android instances, this cannot be done as running multiple system services, each directly accessing devices, would cause conflicts. Alternatively, running multiple system services

Service	Device(s)
AudioFlinger	Microphone, Speakers
CameraService	Camera
LocationManagerService	GPS
SensorService	Motion, Environmental Sensors

Table 3.1: Listing of device container services.

would require an additional mechanism injected below system services in the middle of a complex device stack to somehow multiplex their access to hardware devices, which would be a difficult challenge.

To solve this problem, AnDrone introduces a *device container*, a special container running a minimal Android instance with direct access to hardware devices to run Android's device services. Only a single set of system services are run, just like a vanilla Android instance, and AnDrone leverages the multiplexing functionality in system services to support access to system services and their underlying devices from multiple virtual drone containers. System services are centralized in the device container and removed from all virtual drone containers. AnDrone then makes these services available in all virtual drone containers in place of their own.

To allow virtual drones to use the device services running in the device container, those services need to be registered with each virtual drone's ServiceManager so that the respective ServiceManager can provide a reference to the desired device service when requested by an app. To support this cross-container service registration, we add a new ioctl to the Binder driver, *PUBLISH_TO_ALL_NS*, callable only by the device container for security. Figure 3.6 shows how the ioctl is used. When the device container's ServiceManager receives a new service registration request, it checks to see if the service name is in a pre-specified list of services that are to be shared as shown in Table 3.1. If the service name is in the list, the ServiceManager calls this ioctl to publish it in all running virtual drone containers. The ioctl then takes the service name and handle passed to it and checks all other containers for existing ServiceManagers. The presence of a ServiceManager indicates that the container is a virtual drone running Android Things. The ioctl then makes its own registration call to these existing ServiceManagers with the provided name and

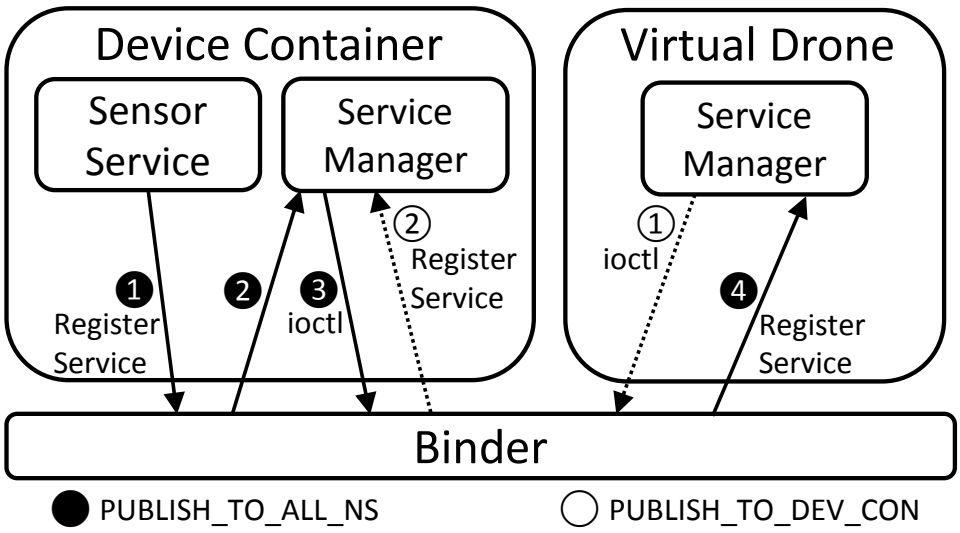

Figure 3.6: Device container service publishing process.

handle, thus registering the device container's service inside the virtual drone container. The same process will be performed in the future for any newly created virtual drone containers. AnDrone also disables the equivalent device services inside the virtual drone containers from starting by modifying init files and Android's *SystemServer*, a process responsible for starting many services. When an app in any virtual drone container asks for a reference to one of the shared services, the respective ServiceManager will return a reference to the single service running inside the device container. Once the app has the reference to the shared service, it can communicate with it via Binder the same as if it were in its own container. All communication with the device services listed in Table 3.1 is fully encapsulated in Binder messages or by using a file descriptor shared via a Binder message.

Although device services are now available in virtual drone containers, a service must also allow apps to use it. In Android, a service asks the ActivityManager if the calling app has permission to use it. This means that a device service will ask the device container's ActivityManager for permission rather than the calling container's ActivityManager, which will be problematic because the device container's ActivityManager will not be aware of the permissions for apps running outside of the device container. To address this problem, we add one more new ioctl to the Binder driver, *PUBLISH_TO_DEV_CON*. We modify each container's ServiceManager to call this ioctl when its

respective ActivityManager registers itself with the ServiceManager, as shown in Figure 3.6. The ioctl appends the ActivityManager service name with the container identifier and registers it with the device container's ServiceManager. We then modify Android's native and Java *checkPermission()* functions in the device container to request the calling container's ActivityManager from the device container's ServiceManager, as identified by the modified service name. To allow services to identify the calling container so that it can perform this check, we make a small modification to Binder to include the calling process' container identifier in its transaction data structure alongside the existing calling process' PID and EUID.

AnDrone's device container model introduces minor additional risk by sharing several services among all of the containers, breaking isolation for those specific services. We deem this an acceptable trade-off as these services are already meant to be used by untrusted apps and thus are already hardened. Billions of active Android devices help demonstrate how secure these already are. Note that our approach does not add any security risk beyond existing Android configurations which share device services among apps. Compared to a standard Android environment with dozens of shared services, AnDrone's shared services represent an order of magnitude reduction in attack surface via system service exploitation. Even if a vulnerability were found in one of the shared device services, it does not fully compromise virtual drones or the whole of the device container and would depend on the device service vulnerability. In the worst case, if the flight controller, discussed in Section 3.4.3, is running on shared hardware with the virtual drones and the GPS or SensorService are compromised, stability and control of the flight can be compromised by the attacker. However, these two services are relatively simple compared to other system services and to the best of our knowledge, and after a review of Android's Common Vulnerabilities and Exposures (CVEs), have never had significant security vulnerabilities discovered in them. Section 3.4.3 discusses how this additional risk to flight control can be mitigated.

3.4.3 Flight Container

A drone's flight is controlled via a flight controller. The flight controller is commonly a native Linux daemon running on the drone itself that is responsible for both stabilizing the drone and accepting commands for maneuvering it. Communication with the flight controller commonly takes place via the Micro Air Vehicle Link (MAVLink) protocol, allowing a ground station or app to have full control of the drone via any underlying medium. To support AnDrone's usage model, we must be able to multiplex the flight controller among virtual drones as well as between virtual drones and the cloud-based flight planner.

To address this problem. AnDrone introduces a flight container for running the flight controller in its own standard Linux container, isolating and prioritizing it over virtual drones due to its mission critical importance. We leverage and modify MAVProxy [104], a portable, minimalist ground control station with MAVLink proxying capabilities, to allow multiple clients to connect to the flight controller. MAVProxy acts as an intermediary between clients and the flight controller, which provides an indirection mechanism to virtualize the flight controller. AnDrone uses MAVProxy to give the cloud-based flight planner full native access to the flight controller, but presents each virtual drone with its own virtual flight controller (VFC) to control the degree of flight control allowed. MAVProxy provides a standard unrestricted flight controller connection for the flight planner and service provider to use, and a VFC connection for each virtual drone which restricts the flight control commands that will be accepted and presents a virtualized view of the drone that differs from that of the physical drone. The extent of the restricted commands is configurable via a whitelist of MAVLink commands available as a number of preconfigured whitelist templates which are customizable by the service provider. The most restrictive template available will only allow the drone to operate in guided mode wherein only a desired GPS position may be given. The least restrictive template allows for full control of the drone so long as it remains within the geofence.

A virtual drone can connect to its VFC at anytime throughout a flight, but until a virtual drone's waypoint is reached, the VFC presents a view of their drone as idle on the ground at the waypoint

to indicate it is inactive and declines any commands sent to it. As the real drone approaches a waypoint, the virtual drone presented automatically takes off to meet the physical drone's position. Once the real drone's position is met, the virtual flight controller begins to accept commands. The commands sent to it from this point on will control the physical drone, but the drone is both geofenced and the commands that the VFC will accept are restricted. The exception to this virtualized view is if the virtual drone has continuous access to devices while operating between its waypoints. To prevent a discrepancy between the view of the drone and device readings, the actual drone's position is given, but commands are still declined until a waypoint is reached. Once the virtual drone is finished with flight control, or is forced to finish, the VFC again refuses to accept commands and presents the drone as landing, where it stays for the remainder of the flight. Meanwhile the physical drone is piloted on to the next waypoint.

MAVLink and flight controllers already support containing drones with geofences, but the action taken when the geofence is breached is to perform a failsafe landing. For AnDrone, this behavior is undesired as the flight must continue so that other virtual drones may operate and eventually return to base. We augment the geofence support such that a breach causes the following steps to be performed: inform the virtual drone of the breach, disable commands on the VFC connection, guide the drone back inside the geofence, and switch it into loiter mode to hold its current position. Flight control is then returned to the virtual drone. With this approach, geofence breaches can be safely handled without interruption to the overall flight.

To run the flight container on the same hardware as the virtual drones, the flight controller must also have access to hardware devices, such as the GPS, which are controlled by and must be accessed via the device container, just like any other virtual drone. However, the device container provides Android-based service interfaces, which are not supported by native Linux. AnDrone introduces additional hardware abstraction layer (HAL) support to the flight container to provide a Binder based bridge between the controller and the device container's device services. Adding HAL support for sensor devices, e.g. barometer, is straightforward since sensor access is supported via the Android Native Development Kit (NDK). However, the NDK does not provide access to

GPS, so a native interface for Android's LocationManagerService had to be created.

Although the flight stack is isolated from virtual drones by containerization, it is still vulnerable to kernel-level faults and vulnerabilities. When sharing hardware with the flight controller, a bug or intentional kernel crash can result in loss of control of the drone. This potential risk is not unique to a shared environment and the risk of a kernel crash is typically handled through additional failsafe mechanisms. For example, the Emlid Navio2 daughterboard includes a failsafe in the on-board microcontroller [105]. This risk can be removed by running the flight controller on separate hardware if desired.

3.4.4 Virtual Drone Controller

To manage virtual drones, AnDrone provides a Virtual Drone Controller (VDC). The VDC is a daemon running natively on the host OS of the physical drone responsible for managing virtual drone containers. Prior to each flight, the flight planner sends the VDC the virtual drone definitions assigned to it. The VDC creates containers for each virtual drone to run as, or if resuming a previous virtual drone flight, obtains the existing virtual drone from the VDR. Once a flight is complete, if a virtual drone is unable to complete its task prior to exhausting its allotted energy or must be interrupted due to unpredictable reasons such as inclement weather, the VDC is responsible for storing the virtual drone, including its updated container image, in the VDR at the end of a flight so that it may be resumed on a later flight. Although checkpoint-based migration is likely feasible for virtual drones [30, 33, 12], AnDrone simply leverages the existing Android activity lifecycle to facilitate saving and restoring the state of virtual drones so they can be migrated between physical drones. Android apps are informed when they are about to be terminated and allowed to save their current state via the *onSaveInstanceState()* callback. The apps can then use this saved state when starting once again to restore themselves as they were prior to being terminated. All AnDrone apps are expected to support this standard Android functionality. A virtual drone's state can then safely be saved offline as part of its disk image.

The VDC also manages virtual drone device access by verifying whether or not a virtual drone

is allowed access to a device throughout a flight. This is done by extending Android's service permission model so that the *checkPermission()* function called when a device service queries the ActivityManager, as discussed in Section 3.4.2, also queries the VDC. The VDC informs the device service if the calling virtual drone container has permission to use the requested device, as defined by its virtual drone definition. The flight planner notifies the VDC throughout the flight once virtual drone waypoints are reached so the VDC can update its device access restrictions. Unlike Android's service permission model which only checks permissions when an app first asks to use a device then allows the app to retain the permissions, AnDrone must be able to revoke permissions of an app that is actively accessing a device, e.g. when leaving a waypoint. To avoid substantial changes to device services to support permission revocation, AnDrone provides this functionality by asking apps to voluntarily disable device access. As discussed in Section 3.5, AnDrone apps make use of an AnDrone SDK and are expected to disable device access upon being informed that they are no longer accessible via the AnDrone SDK. Since apps may choose to ignore the permission revocation notification, the VDC enforces this by asking each device service if there are any processes from the given virtual drone still accessing a device after notification, in which case the VDC terminates those processes. In a similar manner, the VDC is queried by the flight container to determine if a virtual drone has permission to control the flight.

3.5 AnDrone Apps

AnDrone apps are standard Android apps that are written just like any other Android app. However, AnDrone apps also require the ability to interact with AnDrone to know about events specific to AnDrone, such as when they have arrived at a waypoint and when they are finished at a waypoint. AnDrone provides this functionality with a simple AnDrone SDK that apps can use. Figure 3.7 lists the AnDrone SDK methods.

A key component of this SDK is the *WaypointListener* callback class, as shown in Figure 3.8. Apps create an instance of this class and register it with the AnDrone SDK method *registerWaypointListener()* listed in Figure 3.7. Once registered, the app can be notified of various AnDrone

```
void registerWaypointListener(WaypointListener l);
void waypointCompleted();
InetAddress getFlightControllerIP();
void markFileForUser(String path);
int getAllottedEnergyLeft();
int getAllottedTimeLeft();
```

Figure 3.7: AnDrone SDK methods.

```
abstract class WaypointListener {
     waypointActive(Waypoint waypoint);
     waypointInactive(Waypoint waypoint);
     lowEnergyWarning(int remaining);
     lowTimeWarning(int remaining);
     geofenceBreached();
     suspendContinuousDevices();
     resumeContinuousDevices();
}
```

Figure 3.8: Simplified WaypointListener class definition.

related events. An app is notified upon arriving at a waypoint via the *waypointActive()* callback. After receiving this callback, the app knows it is now at the given waypoint, has access to flight control and other waypoint-specific devices it requested, and is free to perform its desired task. Upon leaving a waypoint, either voluntarily or because the maximum time or energy allocation allowed for the virtual drone has been reached, an app is notified via the *waypointInactive()* callback, indicating flight control and waypoint-specific device access is about to be removed and the drone is moving on. The *WaypointListener* also provides callbacks informing apps if their virtual drone is running low on its allotted time or energy allocation via *lowEnergyWarning()* and *lowTimeWarning()*. If the geofence is breached, the app is informed via the *geofenceBreached()* callback, and the app is informed when the virtual drone regains control of the physical drone by a subsequent *waypointActive()* callback. *suspendContinuousDevices()* is called when approaching another party's virtual drone waypoint, indicating that access to devices must be suspended until the other party is finished at their waypoint, as indicated by a call to *resumeContinuousDevices()*. *waypointCompleted()*, listed in Figure 3.7, is called by an app to indicate it has finished its task at a waypoint.

Figure 3.7 lists four additional methods for AnDrone apps. *getFlightControllerIP()* is used to facilitate connecting to the virtual flight controller. *markFileForUser()* is used to indicate files that should be made available to the user in cloud storage after the flight. The final two *getAllotted* functions allow the app to obtain the remaining energy and time allotted for the virtual drone. For advanced end users, who may not be using an app, AnDrone's SDK functionality is also made available to them via a command line utility.

In addition to using the AnDrone SDK, every AnDrone app must include an XML manifest file, similar to the existing Android XML manifest file, indicating the requested device permissions and any arguments it expects from users. The AnDrone manifest is used by the AnDrone portal and flight planner to provide information needed as part of ordering a virtual drone and flight planning. The AnDrone portal reads app arguments from the AnDrone manifest so it knows what arguments an app requires from the user when ordering the virtual drone and prompts the user for these values as part of the ordering process. The AnDrone flight planner needs to know which devices are needed by apps in a virtual drone so it can avoid device access conflicts among virtual drones and control access to devices during flight. Device permission requests are declared in the AnDrone manifest much like existing Android permissions in the Android manifest. A `<uses-permission>` tag is used to specify the `name` and `type` of access requested. `type` can be either that of *waypoint* for devices that only need to be accessed at task waypoints, or *continuous* for access to devices while also between waypoints. Arguments the app requires from the user are declared with an `<argument>` tag, specifying a `name`, `type` of argument, and if the argument is `required`.

3.6 Evaluation

We have implemented an AnDrone prototype and evaluated it in the context of Linux-based drone quadcopter hardware shown in Figure 3.9. The quadcopter uses a DJI Flame Wheel F450 Air Frame [106], equipped with four T-Motor MN2213 950Kv motors [107] with 9.5" propellers attached, and four SimonK 30A electronic speed control units to control the speed of the motors

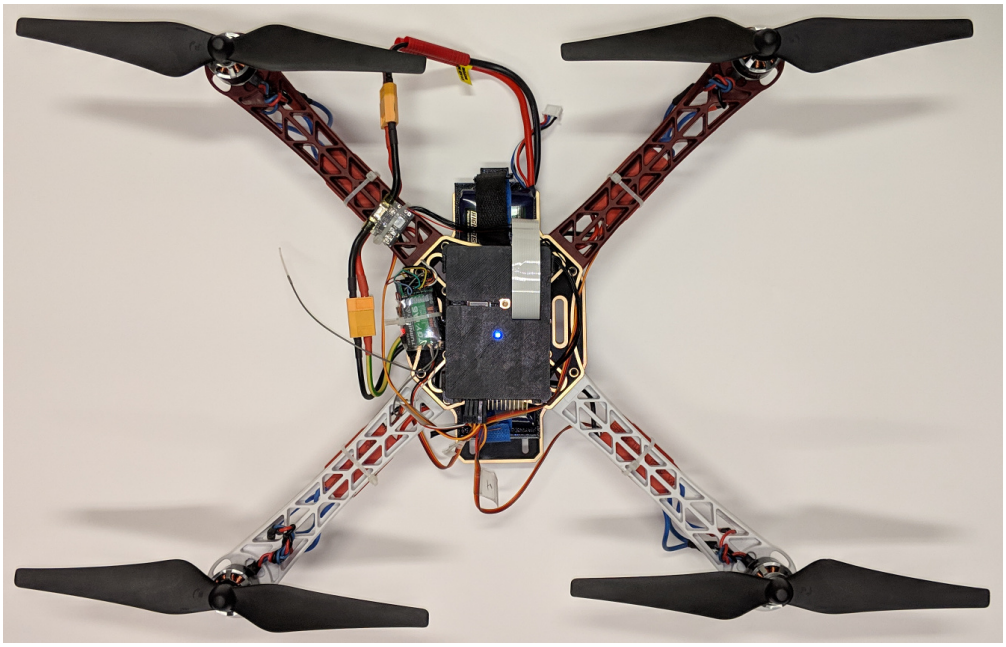

Figure 3.9: AnDrone quadcopter prototype.

mounted beneath the frame. The drone is controlled by a Raspberry Pi 3 Model B [86] (Broadcom BCM2837, 4x Cortex-A53 1.2 GHz CPU, 1GB RAM) single-board computer (SBC) with an attached Emlid Navio2 [87] daughterboard drone controller, Raspberry Pi Camera Module v2 [108], and a SanDisk Extreme 16GB microSDHC card for storage. The SBC is mounted in the center of the Air Frame and the entire drone is powered by a Turnigy 5000mAh 3S battery [109] mounted underneath the SBC. The SBC runs Raspbian [110] Stretch, the official Linux distribution for Raspberry Pi, as the host OS, Android Things v1.0.3 in the virtual drone and device containers, and Alpine Linux [111] v3.7 in the flight container supporting the ArduPilot Copter [112] v3.4.4 flight controller.

3.6.1 Runtime Overhead

We first evaluate the performance of AnDrone when running multiple virtual drones with various workloads. The first workload we used was the popular Android PassMark PerformanceTest benchmark [113], which is commonly used to measure multi-threaded CPU, disk, and memory performance of Android systems. PassMark also has 2D and 3D graphics benchmarks, but we did not

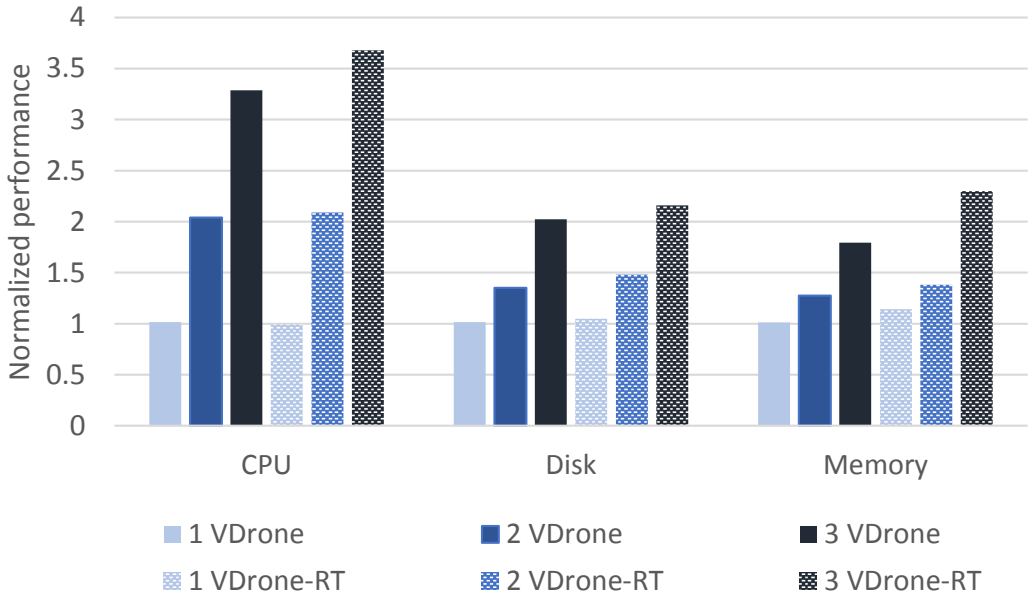

Figure 3.10: AnDrone runtime overhead.

run those as Android Things does not have hardware accelerated GPU support. To measure how runtime overhead is affected by the number of virtual drones running, we ran PassMark in each virtual drone simultaneously with different numbers of virtual drones running. Docker container resource controls were not used. To show the impact of different levels of kernel preemptibility, we measured performance for both the AnDrone default kernel configuration with PREEMPT_RT support enabled versus the minimally accepted real-time support used by Navio2's default kernel configuration with only PREEMPT support enabled. PREEMPT_RT allows the kernel to be almost fully preemptible while PREEMPT disallows kernel preempt when local interrupts are disabled, the latter potentially incurring higher latencies. We normalized PassMark performance compared to running a single instance of PassMark using stock Android Things natively on the system without AnDrone, which does not have PREEMPT_RT or PREEMPT enabled. Running multiple PassMark instances simultaneously is only made possible through virtual drones so only a comparison with a single instance on stock can be made.

Figure 3.10 shows PassMark results normalized to the performance of stock Android Things running a single PassMark instance; lower is better. Results are shown for running with one, two, or three virtual drones in total; three virtual drones means that all three were simultaneously

running the individual PassMark tests. Three virtual drones running simultaneously was the maximum our prototype could support due to memory constraints. We expect future, more powerful drones will be able to support more virtual drones. PREEMPT_RT results are indicated by the "-RT" postfix, while the other results are for just enabling PREEMPT. With a single virtual drone running, CPU, disk, and memory performance remained relatively constant with at most 1.5% overhead, demonstrating the minimal performance overhead of virtual drones. CPU performance shows roughly a linear decrease in performance with a linear increase in the number virtual drones running PassMark, indicating that runtime overhead does not increase significantly with more virtual drones. With three virtual drones, the PREEMPT_RT kernel performed somewhat worse than the PREEMPT kernel, indicating some cost associated with greater kernel preemptibility and more tasks running. On the other hand, disk and memory performance did not decrease as much with an increase in the number of virtual drones running PassMark. Disk performance overhead with three virtual drones was roughly 2x and 2.2x for the PREEMPT and PREEMPT_RT kernels, respectively. Memory performance overhead with three virtual drones was roughly 1.8x and 2.3x for the PREEMPT and PREEMPT_RT kernels, respectively. In practice, we expect that more realistic apps will experience less performance slowdowns as they benefit from multiplexing more variable resource demands.

3.6.2 Real-time Latency

To demonstrate that AnDrone can provide the flight controller, ArduPilot, with sufficient real-time latency guarantees in the presence of various workloads, we ran the commonly used latency benchmark, cyclictest [114], and configured it to run in the flight container in the same way as AnDrone runs ArduPilot by locking all memory allocations and assigning its thread the highest real-time priority. We ran three different workloads at the same time as cyclictest and configured cyclictest to run for 100 million loops to provide sufficient samples to have a high confidence in encountering worst case latencies. First, we ran cyclictest on an otherwise idle system to measure baseline performance. Second, to cause latencies an AnDrone environment is likely to encounter

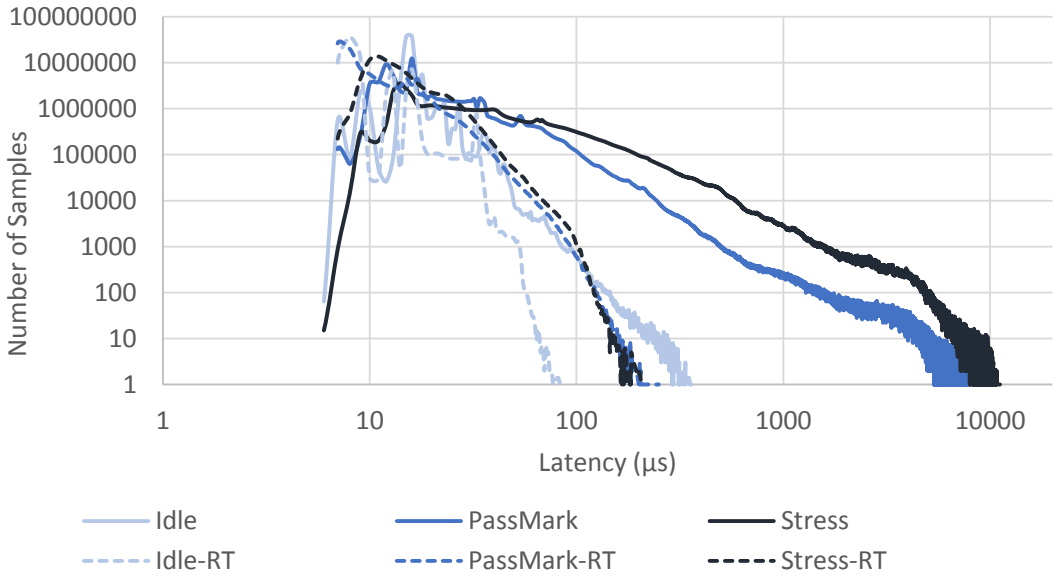

Figure 3.11: AnDrone real-time latency.

under heavy load, we ran cyclictest with three virtual drones running, one idle, one running PassMark continuously in a loop, and one continuously running the iperf [115] network throughput test to stress the system and generate interrupts. Docker container resource controls were not used. Finally, to generate an even worse case latency scenario, we ran cyclictest while stressing all aspects of the system with the stress [116] workload generator to strain CPU, memory, I/O, and disk subsystems, and iperf to strain the network subsystem, both running natively on the host. For both iperf scenarios, iperf was connected over Gigabit Ethernet via a network switch to a Lenovo Thinkpad T540p acting as the iperf server. Stress was configured to run with four CPU worker processes, two I/O worker processes, two memory worker processes, and two disk worker processes. We performed all cyclictests on both the PREEMPT and PREEMPT_RT enabled kernels to compare their performance.

Figure 3.11 shows the latency of each cyclictest measurement for these three workloads and two kernel configurations. PREEMPT_RT results are indicated by the "-RT" postfix, while the other results are for just enabling PREEMPT. The PREEMPT idle, PassMark, and stress scenarios exhibited maximum latencies of 1,307µs, 14,513µs, and 17,819µs and average latencies of 17µs, 44µs, and 162µs, respectively. The PREEMPT_RT idle, PassMark, and stress scenarios exhibited

maximum latencies of 103µs, 382µs, and 340µs and average latencies of 10µs, 12µs, and 16µs, respectively. ArduPilot's most demanding real-time requirement is its most frequently run control loop, the *fast loop*. The fast loop processes values from one or more inertial motion units (IMUs) and adjusts the motors to maintain stability and aid in flying the drone. Ardupilot's fast loop runs at 400Hz, requiring real-time latencies below 2500µs to achieve this. The PREEMPT_RT patched kernel demonstrated latencies well within the requirements of ArduPilot, whereas the PREEMPT kernel did occasionally fall short. However, occasionally missing ArduPilot's fast loop deadline will not cause significant stability issues [117]. Given this, and the infrequency with which the PREEMPT kernel failed to meet ArduPilot's requirements, it is likely this kernel configuration is also sufficient for AnDrone.

To further demonstrate that the stability of the drone is not compromised with AnDrone, we operated our drone prototype at a hover and compared its performance while running the idle and PassMark scenarios described above. The AnDrone default PREEMPT_RT kernel was used for these flight tests. We then analyzed logs of each flight using DroneKit's Log Analyzer [118] and compared them using the Attitude Estimate Divergence (AED) analyzer. The AED analyzer evaluates the flight logs and determines if the flight controller's estimated attitude of the drone differs significantly from the canonical drone attitude, indicating instability if the drone's yaw, pitch, or roll diverges more than 5° from the estimates for longer than .5 seconds. Both scenarios were within normal divergence.

3.6.3 Memory Usage

Since available memory is the primary limitation on how many virtual drones can be run, we quantified the amount of memory used by virtual drones. We first measured the memory usage of AnDrone without any containers, then adding just the device and flight containers, then starting up from one to three virtual drones, the maximum supported by our drone hardware prototype; starting a fourth virtual drone fails due to lack of memory but does not interfere with other virtual drones already running. Each virtual drone was idling on its app launcher screen. Figure 3.12 shows the

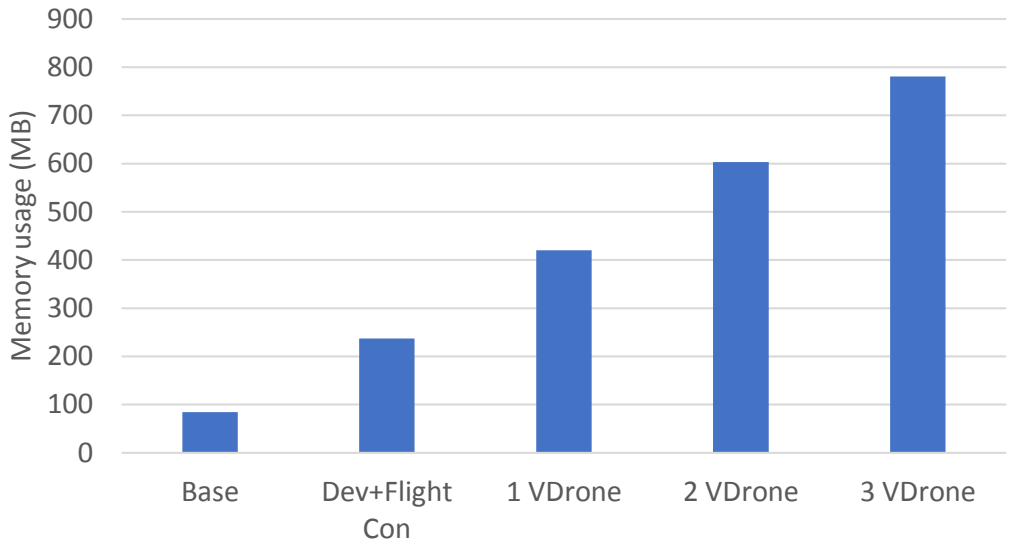

Figure 3.12: AnDrone memory usage.

memory usage of AnDrone in these various configurations. The results show that less than 100MB of RAM is needed to run the VDC and host OS, roughly 150MB of additional RAM is needed to run both the device and flight containers in addition to the base system, and approximately 185MB is needed for each virtual drone. Although our prototype does have 1GB of RAM, only 880MB is made available after accounting for peripheral I/O reserved space and RAM allocated to the GPU for camera functionality.

3.6.4 Power Consumption

To demonstrate that AnDrone has a negligible effect on energy usage, we used a Monsoon Power Monitor [119] to measure the power consumption of AnDrone with the drone at rest, normalized to stock Android Things running on the Raspberry Pi idling on its app launcher screen. We measured energy usage using the same system configurations as described in Section 3.6.3 for measuring memory usage. Figure 3.13 shows the energy usage of AnDrone in these various configurations, with all configurations within 3% of stock Android Things. In absolute numbers, while at idle with three virtual drones running, AnDrone consumed approximately 1.7W. We also measured the energy usage when fully stressing the system using the same stress and iperf workloads used for measuring real-time latency as discussed in Section 3.6.2, but the energy usage was

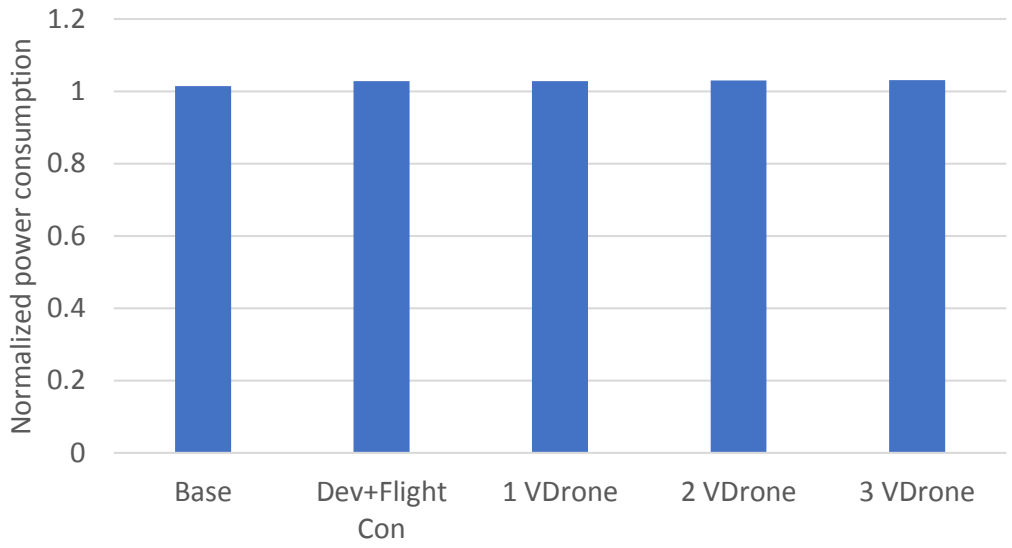

Figure 3.13: AnDrone power consumption.

the same, 3.4W, across both stock Android Things and all AnDrone configurations, so they are omitted from Figure 3.13. Both idle and fully stressed system energy usage are insignificant when compared to the power draw of the rest of the drone. Even consumer-level drone batteries are rated to allow a power draw of well over 100W throughout a 20 minute flight.

3.6.5 Network Performance

Extensive testing and trials by Qualcomm [93] have demonstrated the feasibility of leveraging LTE for real-time control of drones beyond visual line of sight. In a field trial consisting of approximately 1,000 test flights in addition to complementary simulations, Qualcomm demonstrated LTE's ability to support safe drone operation in real-world environments up to 400 feet above the ground with strong signal availability at high altitudes, successful handover and lower frequency of handovers, and comparable coverage to mobile devices on the ground.

To verify their findings with AnDrone and evaluate control of a drone over a cellular network, we used USB tethering to connect our prototype to a Nexus 5X smartphone operating on the T-Mobile cellular network, then conducted various experiments to measure the impact of the cellular network on drone control. First, we qualitatively compared the control responsiveness of flying the drone via a traditional RF-based remote controller versus using the cellular network. For the

latter, we connected a Microsoft Xbox 360 gamepad to a Lenovo Thinkpad T540p laptop running the APM Planner 2 [120] ground station and accessing the Internet via the Columbia University campus WiFi network. We did not notice any significant difference in control responsiveness when using the gamepad to operate the drone over the cellular network versus the RF-based remote controller. To quantify the difference, we set up a testbed environment with the prototype disconnected from drone hardware, but still operating on the cellular network. We then issued roughly 150,000 MAVLink commands over a 12 hour period to the flight controller via the Thinkpad T540p laptop using a wired Verizon Fios Gigabit Connection to the Internet. Although the commands did not succeed since the flight controller was not connected to drone hardware, we could measure the latency between when each command was sent and when the flight controller received it. On average, commands took 70ms to be received with a maximum latency of 356ms and a standard deviation of 7.2ms. 6 packets were lost overall. By comparison, the average RF remote control latency of typical hobby drones ranges from 8ms and 85ms [121, 122].

3.6.6 Multi-waypoint Flight Simulation

To demonstrate AnDrone as a whole under more extensive flight conditions, we used the ArduPilot Software in the Loop (SITL) Simulator [123] to have the AnDrone flight planner deploy virtual drones to various waypoints under simulated flight conditions. For testing with the SITL simulator, we replaced AnDrone's onboard flight controller with a Lenovo Thinkpad T540p laptop running its own ArduPilot flight controller using the SITL simulator software instead of real drone hardware. The virtual drones still communicate with the flight container's MAVProxy via their VFC connections, but MAVProxy communicates with the laptop's simulated flight controller instead of the flight controller inside the flight container. Similarly, the flight planner communicates with the laptop as well. With this setup, we are able to capture core aspects of AnDrone, such as managing device and flight control access among virtual drones, but are not able to capture the effects virtual drones may have on an onboard flight controller on a real-world flight.

We used this SITL simulator setup to perform an AnDrone flight with three virtual drones, one

running an autonomous survey app, another running an interactive app allowing remote control of the drone from a Nexus 5X smartphone, and a third providing direct user access. Upon starting the flight, the VDC created and started the three separate virtual drones from their respective virtual drone definitions. Remote console access was provided for the direct access virtual drone at the start of the flight. The flight planner correctly pathed the drone to the first waypoint for the autonomous survey app virtual drone. Once at the waypoint GPS, camera, and flight control was given to the app, which in turn used the DroneKit [124] API to fly back and forth over a location while recording video as an app might do for surveying a field. After calling the AnDrone *waypointCompleted()* SDK method, the flight planner pathed the drone to the second waypoint for the interactive app virtual drone. The app successfully allowed maneuvering the drone, and an intentional geofence breach was handled as expected. At the next waypoint, control was handed to the direct access virtual drone. The APM Planner base station running on the ThinkPad, as discussed in Section 3.6.5, was used to successfully connect to the virtual drone's VFC. Access to the camera was allowed while at the waypoint whereas previous access attempts were denied. Finally, the drone returned to its base.

3.7 Related Work

Various approaches have been proposed for providing drone services, though few have been implemented and none of them support virtual drones. An IBM patent application [125] describes an autonomous drone service system to allow users to order specific drone services that will be performed either autonomously or by a ground-based pilot. Unlike AnDrone, users cannot perform any service beyond those offered by the service provider and are not given flight control or access to a drone. Additionally, such a service requires both a drone and a pilot for each order, increasing the cost of such services.

UAV as a Service (UAVaaS) [126] is a proposed framework enabling users to connect to a cloud-based service to use a drone. Unlike AnDrone, UAVaaS does not give direct access to drones, but instead supplies users with cloud-based APIs allowing for control of a drone and access to its

device data. Users connect to a cloud-based UAVaaS Coordinator service, which in turn connects to a drone. Because of this, all existing drone and device code is incompatible and new apps and services must be written explicitly for UAVaaS. Additionally, as users interact with the drone through a cloud intermediary and cannot directly run anything on the drone, applications, such as autonomous control apps, may not have sufficient reliability, latency, and bandwidth guarantees to function. For example, if faced with intermittent networking issues, an autonomous AnDrone app will remain unaffected while it conducts its task, whereas any remotely running app will likely have to abort the flight.

Dronemap Planner [127] provides access to drones for developers through web services. To achieve this, a cloud-based MAVLink to WebSocket proxy is created allowing web-based control over a drone's flight. Other than MAVLink proxying, no additional drone functionality is offered and no other device access is possible. Unlike AnDrone, drones can only be navigated and exclusive, unrestricted control is always given to the operator. Any intermittent networking issues will cause the same problems that occur for UAVaaS.

UAV-Cloud [128, 129] provides middleware that facilitates developing collaborative drone apps. Various drone resources such as sensors are made available as cloud-based RESTful APIs for collaborative drone apps to leverage. Beyond defining these APIs, little is actually implemented. Unlike UAV-Cloud, AnDrone's goal is not to abstract away aspects of a drone for collaborative apps, but to offer a complete drone-as-a-service solution making drones accessible in the cloud.

Fly4SmartCity [130, 131] is an emergency-management service that offers aerial support to people in need. A user, e.g. a citizen, requiring aerial support may request it via a mobile app. A cloud service then dispatches a drone and privileged users, e.g. police officers, are given web-based access to its camera feed. Unlike AnDrone, Fly4SmartCity is limited to offering users web-based camera access and is primarily focused on path planning of city-based drones.

FarmBeats [132] is an IoT platform for agriculture that enables data collection from sensors, cameras, and drones to facilitate farm analytics. As part of this, FarmBeats leverages drones connected via an IoT base station, which is in turn connected to a farmer's internet connection via

a radio link. These drones map fields, monitor crop canopy, and check for anomalies. Unlike AnDrone, the primary focus of FarmBeats is leveraging user-owned drones, and other sensors, to supply analytical data to the cloud.

Similar to AnDrone's goal of fully leveraging flight time, Galois, Inc. has investigated leveraging the spare capacity of flight controller hardware by porting FreeRTOS to run in a Xen VM on ARM Cortex A15 based devices [133, 134] so the VM can be used to run SMACCMPilot [135]. Their approach suffers from the difficulty of exposing the vast array of sensor devices available to the VM. There has been no indication that full support for running SMACCMPilot inside a Xen VM has been achieved. More generally, there exists ongoing projects for adding support to both the Xen [2] and KVM [136] for running VMs with real-time requirements. This could potentially allow for stronger isolation between a flight controller and third-party VMs. However, challenges for such a system exist given both the lack of hardware virtualization support with drone hardware and the difficulty in supporting and passing low-latency access to the numerous and diverse sensor devices to VMs for use by real-time apps.

Like AnDrone, Cells [11] leverages containers to support multiple Android instances on the same hardware, though Cells focuses on smartphones and tablets where graphics support is of key importance. Cells introduces a new type of namespace for devices to accomplish this, but assumes a more simplified use case model appropriate for smartphones and tablets. With Cells, the user interacts with one foreground Android instance at a time, while background Android instances mostly do not need hardware device access. In contrast, AnDrone must allow for any virtual drone requiring access to physical hardware devices at any time, so more fine grained access control is needed than what is supported by Cells. Device namespace support can be tedious and error prone as new devices may require kernel driver modifications. Because device namespaces require contextual knowledge of how a device operates to implement support for a given device, supporting opaque peripheral devices without specific kernel drivers can be a challenge. These devices maintain their context in userspace, communicate through userspace over buses like Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I2C), and the kernel only sees raw reads

and writes. In contrast, AnDrone's device container design has no such limitation as it operates at the system service level, and requires no per-device support, significantly reducing the effort required to support multiple Android instances on new platforms.

3.8 Summary

We have designed, implemented, and evaluated AnDrone, the first drone-as-a-service solution making drones fully accessible in the cloud. AnDrone introduces virtual drones for the first time, enabling drone tasks for multiple users to be consolidated on the same physical drone and performed during the same flight. Users can create and configure Android Things virtual drones in the cloud with various apps and services of interest, leveraging a large existing base of Android apps, developers, and resources, which are then safely deployed and multiplexed on real drone hardware. AnDrone achieves this by pairing a cloud service with a lightweight virtualization architecture that introduces a new device and flight container design for multiplexing drone hardware and managing virtual drone device access, including providing virtual drone geofenced flight control. Multiple variants of Linux can be run at the same time, including Android Things virtual drones together with a real-time Linux flight controller.

We have implemented an AnDrone prototype and used it together with drone quadcopter hardware based on the Raspberry Pi 3 Model B and Emlid Navio2 daughterboard. Our experimental results demonstrate runtime performance overhead of less than 1.5% for a single virtual drone, the ability to run multiple virtual drones for the first time with performance that scales linearly with workload without any significant increase in energy costs, and sufficient low-latency performance for real-time flight controllers while multiplexing multiple virtual drones. Real-world and simulator-based flight demonstrations show that virtual drones can run simultaneously without compromising the stability and safety of the drone.

Chapter 4: BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems

4.1 Introduction

Containers are widely deployed to package, isolate, and multiplex applications on shared computing infrastructure. They are increasingly used in lieu of hypervisor-based virtual machines (VMs) because of their faster startup time, lower resource footprint, and better I/O performance [5, 21, 49, 13]. Popular container mechanisms such as Linux containers rely on a commodity operating system (OS) to enforce their security guarantees. However, commodity OSes such as Linux are huge, complex, and imperfect pieces of software. Attackers that successfully exploit OS vulnerabilities may gain unfettered access to container data, compromising the confidentiality and integrity of containers—an undesirable outcome for both computing service providers and their users.

Modern systems incorporate hardware security mechanisms to protect applications from an untrusted OS, such as Intel Software Guard Extensions (SGX) [137] and Arm TrustZone [138], but they require rewriting applications and may impose high overhead to use OS services. Some approaches have built on these mechanisms to protect unmodified applications [139] or containers [140]. Unfortunately, they suffer from high overhead, incomplete and limited functionality, and massively increase the trusted computing base (TCB) through a library OS or runtime system, potentially trading one large vulnerable TCB for another.

As an alternative, hypervisors have been augmented with additional mechanisms to protect applications from an untrusted OS [141, 142, 143, 144, 145]. This incurs the performance overhead of hypervisor-based virtualization, which containers were designed to avoid. The TCB of these systems is significant, in some cases including an additional commodity host OS, providing addi-

tional vulnerabilities to exploit to compromise applications. Theoretically, these approaches could be applied to microhypervisors [146, 147] with smaller TCBs. Unfortunately, microhypervisors still inherit the complexity of hypervisor-based virtualization, including virtualizing and managing hardware resources. The reduction in TCB is achieved through a much reduced feature set and limited hardware support, making their deployment difficult in practice.

To address this problem, we have created BlackBox, a new container architecture that provides fine-grain protection of application data confidentiality and integrity without the need to trust the OS. BlackBox introduces a *container security monitor (CSM)*, a new mechanism that leverages existing hardware features to enforce container security guarantees in a small trusted computing base (TCB) in lieu of the OS. The monitor creates protected physical address spaces (PPASes) for each container to enforce physical memory access controls, but provides no virtualization of hardware resources. Physical memory mapped to a container's PPAS is not accessible outside the PPAS, providing physical memory isolation among containers and the OS. Since container private data in physical memory only resides on pages in its own PPAS, its confidentiality and integrity is protected from the OS and other containers.

The CSM repurposes existing hardware virtualization support to run at a higher privilege level and create PPASes, but is itself not a hypervisor and does not virtualize hardware. Instead, the OS continues to access devices directly and remains responsible for allocating resources. This enables the CSM to be minimalistic and simple while remaining performant. By supporting containers directly without virtualization, no additional guest OS or complex runtime needs to run within the secured execution environment, minimizing the TCB within the container itself.

Applications running in BlackBox containers do not need to be modified and can make use of OS services via system calls, with the added benefit of their data being protected from the OS. The monitor interposes on all transitions between containers and the OS, clearing container private data in CPU registers and switching PPASes as needed. The only time in which any container data in memory is made available to the OS is as system call arguments, which only the monitor itself can provide by copying the arguments between container PPASes and the OS. The monitor is

aware of system call semantics and encrypts system call arguments as needed before passing them to the OS, such as for interprocess communication between processes, protecting container private data in system call arguments from the OS. Given the growing use of end-to-end encryption for I/O security [148], in part due to the Snowden leaks [149], the monitor relies on applications to encrypt their own I/O data to simplify its design. Once a system call completes and before allowing a process to return to its container, the monitor checks the CPU state to authenticate the process before switching the CPU back to using the container's PPAS.

In addition to ensuring a container's CPU and memory state is not accessible outside the container, BlackBox protects against malicious code running inside containers. Only trusted binaries, which are signed and encrypted, can run in BlackBox containers. The monitor is required to decrypt the binaries, so they can only run within BlackBox containers with monitor supervision. The monitor authenticates the binaries before they can run, so untrusted binaries cannot run in BlackBox containers. It also guards against memory-related Iago attacks, attacks that maliciously manipulate virtual and physical memory mappings, that could induce arbitrary code execution in a process in a container by preventing virtual or physical memory allocations that could overwrite a process's stack.

We have implemented BlackBox on Arm hardware, given Arm's growing use in personal computers and cloud computing infrastructure along with its dominance on mobile and embedded systems. We leverage Arm hardware virtualization support by repurposing Arm's EL2 privilege level and nested paging, originally designed for running hypervisors, to enforce separation of PPASes. Unlike x86 root operation for running hypervisors, Arm EL2 has its own hardware system state. This minimizes the cost of trapping to the monitor running in EL2 when calling and returning from system calls because system state does not have to be saved and restored on each trap. We show that BlackBox can support widely-used Linux containers with only modest modifications to the Linux kernel, and inherits support for a broad range of Arm hardware from the OS. The implementation has a TCB of less than 5K lines of code plus a verified crypto library, orders of magnitude less than commodity OSes and hypervisors. With such a reduced size, the CSM is sig-

nificantly easier for developers to maintain and ensure the correctness of than even just the core virtualization functionality of a hypervisor. We show that BlackBox can provide finer granularity and stronger security guarantees than traditional hypervisor and container architectures with only modest performance overhead for real application workloads.

4.2 Threat Model and Assumptions

Our threat model is primarily concerned with OS vulnerabilities that may be exploited to compromise the confidentiality or integrity of a container's private data. Attacks in scope include compromising the OS or any other software to read or modify private container memory or register state, including by controlling DMA-capable devices, or via memory remapping and aliasing attacks. We assume a container does not voluntarily reveal its own private data whether on purpose or by accident, but attacks from other compromised containers, including confidentiality and integrity attacks, are in scope. Availability attacks by a compromised OS are out of scope. Physical or side-channel attacks [150, 151, 152, 153, 154, 155] are beyond the scope of the paper. Opportunities for side-channel attacks are greater in BlackBox than in systems that isolate at a lower level, e.g. VMs. The trust boundary of BlackBox is that of the OS's system call API, enabling adversaries to see some details of OS interactions such as sizes and offsets.

We assume secure key storage is available, such as provided by a Trusted Platform Module (TPM) [156]. We assume the hardware is bug-free and the system is initially benign, allowing signatures and keys to be securely stored before the system is compromised. We assume containers use end-to-end encrypted channels to protect their I/O data [157, 158, 148]. We assume the CSM does not have any vulnerabilities and can thus be trusted; formally verifying its codebase is future work. We assume it is computationally infeasible to perform brute-force attacks on any encrypted container data, and any encrypted communication protocols are assumed to be designed to defend against replay attacks.

4.3 Design

BlackBox enclaves traditional Linux containers to protect the confidentiality and integrity of container data. We refer to a container as being enclaved if BlackBox protects it from the OS. From an application's perspective, using enclaved containers is little different from using traditional containers. Applications do not need to be modified to use enclaved containers and can make use of OS services via system calls. Container management solutions [101, 100] such as Docker [9] can be used to manage enclaved containers. BlackBox is designed to support commodity OSes, though minor OS modifications are needed to use its enclave mechanism, in much the same way that OS modifications are typically required to take advantage of new hardware features. However, BlackBox does not trust the OS and a compromised OS running enclaved containers cannot violate their data confidentiality and integrity.

BlackBox introduces a container security monitor (CSM), as depicted in Figure 4.1, which serves as its TCB. The CSM's only purpose is to protect the confidentiality and integrity of container data in use. It achieves this by performing two main functions, access control and validating OS operations. Its narrow purpose and functionality makes it possible to keep the CSM small and simple, avoiding the complexity of many other trusted system software components. For example, unlike a hypervisor, the CSM does not virtualize or manage hardware resources. It does not maintain virtual hardware such as virtual CPUs or devices, avoiding the need to emulate CPU instructions, interrupts, or devices. Instead, interrupts are delivered directly to the OS and devices are directly managed by the OS's existing drivers. It also does not do CPU scheduling or memory allocation, making no availability guarantees. The CSM can be kept small because it presumes the OS is CSM-aware and relies on the OS for complex functionality such as bootstrapping, CPU scheduling, memory management, file systems, and interrupt and device management.

To enclave containers, the CSM introduces the notion of a *protected physical address space (PPAS)*, an isolated set of physical memory pages accessible only to the assigned owner of the PPAS and the CSM. Each page of physical memory is mapped to at most one PPAS. The CSM

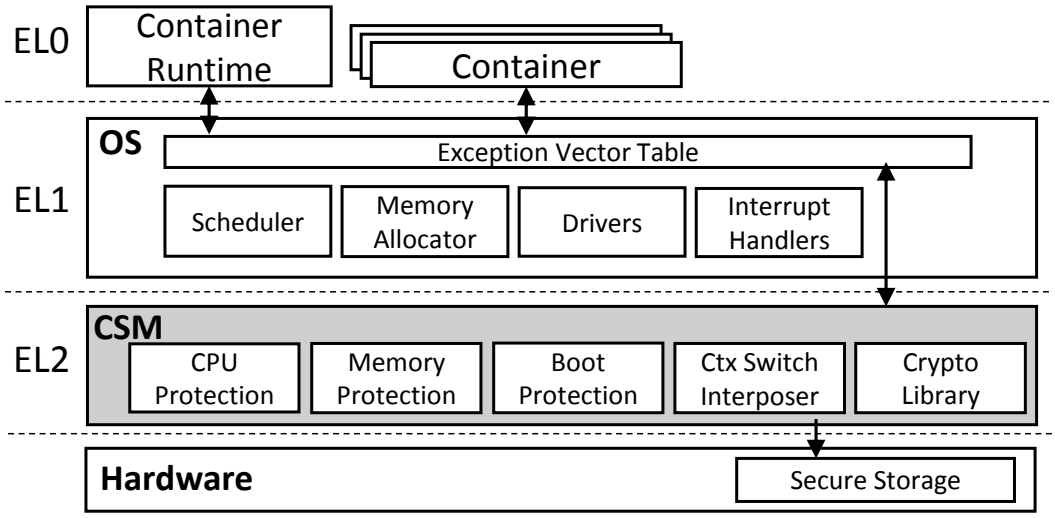

Figure 4.1: BlackBox Architecture.

uses this mechanism to provide memory access control by assigning a separate PPAS to each enclaved container, thereby isolating the physical memory of each container from the OS and any other container. The OS determines what memory is allocated to each PPAS, but cannot access the memory contents of a PPAS. Similarly, a container cannot access a PPAS that it does not own. Memory not assigned to a PPAS, or the CSM, is assigned to and accessible to the OS. The CSM itself can access any memory, including memory assigned to a PPAS. Within a PPAS, addresses for accessing memory are the same as the physical addresses on the machine; physical memory cannot be remapped to a different address in a PPAS. For example, if page number 5 of physical memory is assigned to a PPAS, it will be accessed as page number 5 from within the PPAS. Container private data in memory only resides on pages mapped to its own PPAS, therefore its confidentiality and integrity is protected from the OS and other containers. Section 4.4 describes how BlackBox uses nested page tables to enforce PPASes.

The CSM interposes on all transitions between containers and the OS, namely system calls, interrupts, and exceptions, so that it can ensure that processes and threads, which we collectively refer to as tasks, can only access the PPAS of the container to which they belong when executing within context of the container. The CSM ensures that when a task traps to the OS and switches to running OS kernel code, the task no longer has access to the container's PPAS. Otherwise, the

OS could cause the task to access the container's private data, compromising its confidentiality or integrity. The CSM maintains an *enclaved task array*, an array with information for all tasks running in enclaved containers. When entering the OS, the CSM checks if the calling task is in an enclaved container, in which case it saves to the enclaved task array the CPU registers and the cause of the trap, switches out of the container's PPAS, and clears any CPU registers not needed by the OS. When exiting the OS, the CSM checks the enclaved task array if the running task belongs to an enclaved container, in which case it validates the current CPU context, namely the stack pointer and page table base register, match what was saved in the enclaved task array for the respective task. If they match, the CSM switches to the respective container's PPAS so the task can access its enclaved CPU and memory state. As a result, container private data in CPU registers or memory is not accessible to the OS.

To support OS functionality that traditionally requires access to a task's CPU state and memory, the CSM provides an application binary interface (ABI) for the OS to request services from the CSM. The CSM ABI is shown in Table 4.1. For example, `create_enclave` and `destroy_enclave` are called by the OS in response to requests from a container runtime, such as `runC`[159], to enclave and unenclave containers, respectively. For CSM calls that require dynamically allocated memory, the OS must allocate and pass in the physical address of a large enough region of contiguous memory to perform the respective operation. Otherwise, the call will fail and return the amount of memory required so that the OS can make the call again with the required allocation. For example, `create_enclave` requires the OS to allocate memory to be used for metadata for the enclaved container. Upon success, the allocated memory is assigned to the CSM and no longer accessible to the OS until `destroy_enclave` is called, at which point the memory is assigned back to the OS again.

4.3.1 System Boot and Initialization

BlackBox boots the CSM by relying on Unified Extensible Firmware Interface (UEFI) firmware and its signing infrastructure with a hardware root of trust. The CSM and OS kernel are linked as

CSM Call	Description
`create_enclave`	Create new enclave for a container.
`destroy_enclave`	Destroy enclave of a container.
`protect_vectors`	Validate OS exception vectors.
`alloc_iopgtable`	Allocate I/O device page table.
`free_iopgtable`	Free I/O device page table.
`set_iopt`	Update entry in I/O device page table.
`get_ioaddr`	Get physical address for I/O virtual address.
`enter_os`	Context switch CPU to OS.
`exit_os`	Context switch CPU from OS.
`set_vma`	Update virtual memory areas of a process/thread.
`set_pt`	Update page table entry of a process/thread.
`copy_page`	Copy contents of a page to a container.
`task_clone`	Run new process/thread in a container.
`task_exec`	Run in new address space in a container.
`task_exit`	Exit a process or thread in a container.
`futex_read`	Read the value of a futex in a container.

Table 4.1: BlackBox Container Security Monitor ABI.

a single binary which is cryptographically signed, typically by a cloud provider running BlackBox containers; this is similar to how OS binaries are signed by vendors like Red Hat or Microsoft. The binary is first verified using keys already stored in secure storage, ensuring that only the signed binary can be loaded. To keep the CSM as simple as possible, BlackBox does not implement bootstrapping within the CSM itself, which can require thousands of lines of code to support many systems. Instead, it relies on the OS's bootstrapping code to install the CSM securely at boot time since the OS is initially benign. By relying on commodity OSes such as Linux that already boot on a wide range of systems, this makes it easier for the CSM to support many systems without the burden of manually maintaining and porting its own bootstrapping code for many systems.

At boot time, the OS initially has full control of the system to initialize hardware, and installs the CSM. CSM installation occurs before local storage, network and serial input services are available, so remote attackers cannot compromise its installation. Once installed, the CSM runs at a higher privilege level than the OS and subsequently enables PPASes as needed. A small amount of physical memory is statically assigned to the CSM, and the rest is assigned to the OS. Any attempt to access the CSM's memory except by the CSM itself will trap to the CSM and be rejected. Although the OS's memory is separate from the CSM's, the CSM can access the OS's memory and

can restrict its from modifying its own memory if needed.

The CSM expects the hardware to include an IOMMU to protect against DMA attacks by devices managed by the OS [160]. The CSM retains control of the IOMMU and requires the OS to make CSM calls to update IOMMU page table mappings, which are typically configured by the OS during boot. This ensures that I/O devices can only access memory mapped into the IOMMU page tables managed by the CSM. The OS calls `alloc_iopgtable` during boot to allocate an IOMMU translation unit and its associated page table for a device, and `set_iopt` and to assign physical memory to the device to use for DMA. The CSM ensures that the OS can only assign its own physical memory to the IOMMU page tables, ensuring that DMA attacks cannot be used to compromise CSM or container memory.

4.3.2 Enclaved Container Initialization

To securely initialize an enclaved container, an image that is to be used for such a container must first be processed into a BlackBox container image, using a process similar to how Amazon enclaves are created using Docker images [161]. BlackBox provides a command line tool `build_bb_image`, which can be used by a cloud customer, that takes a Docker image, finds all executable binary files contained within the image, and encrypts the sections containing the code and data used by the code using the public key paired with a trusted private key stored in the secure storage of the host and accessible only by the CSM. These encrypted sections are then hashed and their hash values recorded along with the binaries they belong to. These values are then signed with the private key of the container image's creator whose paired public key is accessible in the secure storage of the host to ensure authenticity and bundled with the container image for later reference during process creation, as described in Section 4.3.3. This ensures the binaries cannot be modified without being detected, or run unless decrypted by the CSM. Other than additional hashes and using encrypted binaries, the BlackBox container image contains nothing different from a traditional Docker image.

To start a container using a BlackBox container image, the container runtime is modified to

execute a simple shim process in place of the container's specified init process. The container runtime passes the shim the path of the init process used by the container along with any arguments and its environment. The shim is also given the signed binary hash information bundled with the container image. The shim process runs a tiny statically linked program that initiates a request to the OS to call the `create_enclave` CSM call before executing the original init process, passing the signed hash information to the CSM as part of the call. Other than the shim process, which exits upon executing the init process, there is no additional code that runs in a BlackBox container beyond vanilla Linux containers. There are no additional libraries and no need for a library OS, avoiding the risks of bloating the TCB of the container itself.

`create_enclave` creates a new enclave using the BlackBox container image and returns with the calling process running in the enclaved container, the return value of the call being the new enclave's identifier. `create_enclave` performs the following steps. First, it creates a new PPAS for the container. Second, it freezes the userspace memory of the calling process so it, and its associated page tables, cannot be directly changed by the OS, then moves all of its pages of physical memory into the container's PPAS so that they are no longer accessible by the OS. Finally, it checks the contents of the loaded shim binary in memory against a known hash to validate the calling process is the expected shim process.

After returning from `create_enclave`, the shim executes the container's init process from within the container. Since the container's init process obtains its executable from the BlackBox container image whose code and data are encrypted, the OS may load it, but cannot actually execute it without the CSM using its private key to decrypt it. Further details on `exec` with encrypted binaries are described in Section 4.3.6. In this way, the OS is incapable of running a BlackBox container image without the CSM. Therefore, if it is running, the CSM must be involved and protecting it. Because the CSM itself is securely booted and enclave code is encrypted and only runnable by the CSM, an unbroken chain of trust is established enabling remote attestation similar to that of other security systems, such as Samsung Knox [162].

The container runtime calls `destroy_enclave` to remove the enclave of a container, which

terminates all running processes and threads within the container to ensure that any container CPU state and memory is cleared and no longer accessible to the OS or any other container before removing the enclave. The container is effectively returned to the same state it was in before `create_enclave` was called.

4.3.3 Enclaved Task Execution

BlackBox supports the full lifecycle of tasks executing in enclaved containers, including their dynamic creation and termination via standard system calls such as `fork`, `clone`, `exec`, and `exit`. This includes tracking which tasks are allowed to execute in which containers. This is achieved by requiring the OS to call a set of CSM calls, `task_clone` on task creation via `fork` and `clone`, `task_exec` when loading a new address space via `exec`, and `task_exit` when a task exits via `exit`. These calls request the CSM to perform various functions related to task execution that the OS is not able to do because it does not have access to task CPU state and memory. If the OS does not make the respective CSM call, the created task and executed binary will simply not run in its enclave and therefore will not have access to its data. These calls update the enclaved task array, the index of which is used as the enclaved task identifier. Each entry in the array includes the enclave identifier of the container in which the task executes, as well as the address of the page table used by the task as discussed earlier.

When a task running in an enclaved container creates a child task via a system call, the OS calls `task_clone` with the enclaved task identifier of the calling task and a flag indicating whether the new task will share the same address space as the caller, as when creating a thread, or have its own copy of the address space of the caller, as when creating a process. In the latter case, new page tables will be allocated for the child task and the CSM will ensure that they match those of the caller's and cannot be directly modified by the OS. The CSM will also confirm that the calling task issued the task creation system call. If all checks pass, the CSM will create a new entry in the enclaved task array with the same enclave identifier as the calling process, and return the array index of the new entry as the identifier for the task. The entry will also contain the address of the

task's page table, which will be the same as the caller's entry if it shares the same address space as the caller.

When the OS runs the child and the task returns from the OS, the OS provides the CSM with the enclaved task's identifier. The CSM then looks up the task in its enclaved task array using this identifier and confirms that the address of the page table stored in the entry matches the address stored in the page table base register of the CPU. If the checks pass, it will then restore CPU state and switch the CPU to the container's PPAS, thereby allowing the task to resume execution in the container. If the OS does not call `task_clone`, then upon exiting the OS, the task's PPAS would not be installed and it would fail to run.

On `exec`, the calling task will replace its existing address space with a new one. The OS calls `task_exec`, which, like `task_clone` for `fork`, creates a new enclaved task entry with a new address space. The difference is that the new address space is validated by ensuring that the new process' stack is set up as expected and the executable binary is signed and in the BlackBox container image, as described in Section 4.3.6. After creating the new enclaved task entry, the original address space is disassociated from the container, scrubbing any memory pages to be returned to the OS and removing them from the container's PPAS.

On `exit`, the OS will call `task_exit` so the CSM can remove the enclaved task entry from the enclaved task array. If an address space has no more tasks in the container, the CSM disassociates it in a similar manner to the `exec` case.

4.3.4 Memory

BlackBox prevents the OS from directly accessing a container's memory, but relies on the OS for memory management, including allocating memory to tasks in the container. This avoids introducing complex memory management code into BlackBox, keeping it small and simple, but means that BlackBox also needs to protect against memory-based Iago attacks [163] by the untrusted OS through manipulation of system call return values. For example, if a process calls `mmap`, it expects to receive an address mapping that does not overlap with any of its existing mappings. If

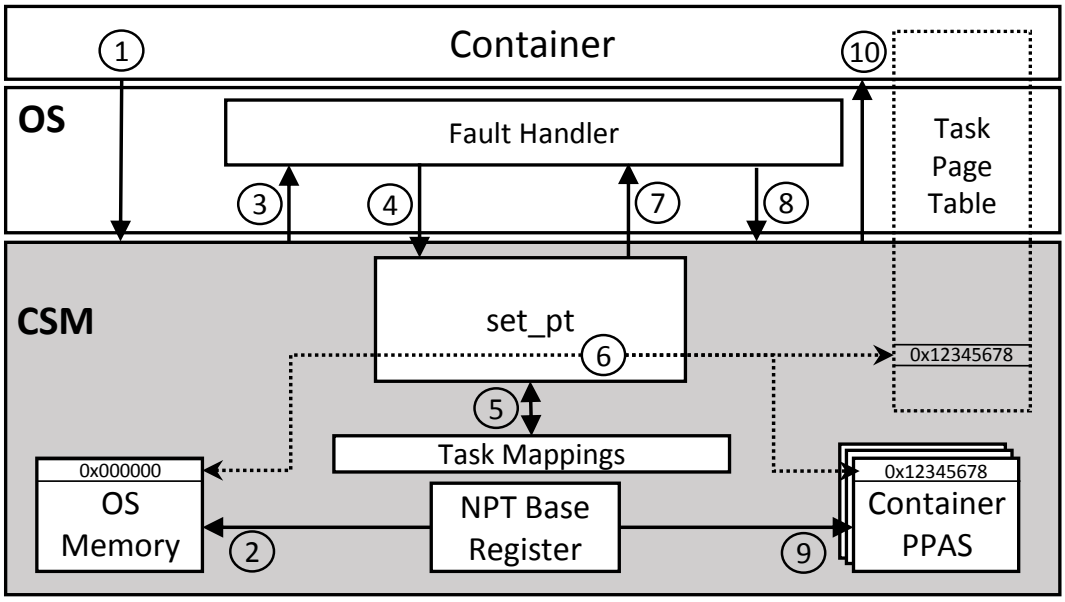

Figure 4.2: BlackBox Page Fault Workflow.

the OS were to return a value overlapping the process's stack, it could manipulate the process into overwriting a return address on its stack through a subsequent `read` with an attacker controlled address, opening the door for return-oriented-programming [164] and return-into-libc [165] attacks. Furthermore, the OS may return an innocuous looking non-overlapping virtual address from `mmap`, but still maliciously map the returned address to the physical page the stack is on.

To rely on the OS for memory management while preventing memory-based Iago attacks, BlackBox protects the container's memory at the application level by preventing the OS from directly updating per process page tables. It instead requires the OS to make requests to the CSM to update process page tables, allowing the CSM to reject updates if the OS behaves incorrectly. Figure 4.2 depicts how a container's page table is updated during a page fault. When a process in a container faults on a page, an exception causes control to transfer to the OS by way of the CSM (steps 1-3). The OS then allocates a page for the process, but instead of updating the process page table directly, it performs a `set_pt` CSM call (step 4). Upon receiving the `set_pt` call, the CSM verifies if the allocation is acceptable (step 5). To do so, the CSM maintains a list of valid mappings for each process. This list is maintained by interposing on system calls that adjust memory mappings. In Linux these calls include `mmap` and `brk`. Prior to writing the page table

entry, the CSM first verifies that the virtual address specified belongs to a valid mapping. If it does not, the update is rejected. Second, the CSM checks if the physical page assigned is already in the container's PPAS and therefore already in use. This can commonly occur innocuously when, e.g., two processes in a container have the same file mapped in their address spaces. However, to prevent the risk of a malicious OS coercing an enclave to overwrite existing memory via a malicious memory allocation, the CSM marks any physical page mapped more than once read only in the container's PPAS, unless it was inherited from a parent as part of process creation in which case it can be trusted. While this is effective at preventing these attacks, the downside is that writes to such memory will trap and need to be handled by BlackBox; for simplicity, BlackBox disallows writable memory-mapped file I/O as it is uncommonly used. Finally, if the virtual address is valid and not mapped to an existing physical page in a container's PPAS, the CSM unmaps the assigned physical page from the OS and maps it into the container's PPAS. The CSM then updates the page table entry on the OS's behalf (step 6). Control is then returned back to the OS (step 7). When returning control back to the process that faulted, the process's container PPAS will be switched to (steps 8-10). Section 4.4 describes further details about this process. The CSM also invalidates TLB entries as needed when it performs page table updates, ensuring that a malicious OS cannot violate a container's PPAS through stale TLB entries.

BlackBox provides support for copy-on-write (CoW) memory, a key optimization commonly used in OSes. The OS traditionally expects to be able to share a page in memory among multiple processes and when a write is attempted, break the CoW by copying the contents of the page to a new page assigned to the process. With BlackBox, the OS does not have the ability to copy container memory though, so the OS instead makes a `copy_page` CSM call to have the CSM perform the CoW break on its behalf. The CSM will check that the source page belongs to the container's PPAS and the destination page is in the OS's memory. If so, it will move the destination page into the container's PPAS and perform the copy.

BlackBox supports the dynamic release of memory back to the OS as tasks adjust their heap, unmap memory regions, and exit, while preserving the privacy and integrity of a container's mem-

ory. As with memory allocation, system calls that can allow for returning of an application's memory, like `munmap` and `_exit` are tracked to maintain an accurate view of a container's memory mappings. During these calls, the OS may attempt to free pages allocated to the process. In doing so, as with memory allocation, it must make use of the `set_pt` CSM call since it cannot update page tables directly. The CSM will then check if the application has made a call to release the specified memory and reject the update if it has not. If the update is valid, the CSM will perform the page table update, and if no longer needed, scrub the page and remove it from the container's PPAS.

While BlackBox ensures that container memory is not accessible to the OS, many OS interactions via system calls expect to use memory buffers that are part of an application's memory to send data to, or receive data from, the OS. BlackBox treats the use of such memory buffers in system calls as implicit directives to declassify the buffers so they can be shared with the OS. To support this declassification while ensuring that a container's PPAS is not accessible to the OS, BlackBox provides a syscall buffer for each task running in an enclaved container that is outside of the container's PPAS and accessible to the OS. When interposing on a system call exception, the CSM replaces references to memory buffers passed in as system call arguments with those to the task's syscall buffer. For buffers that are used to send data to the OS, the data in those buffers is copied to the syscall buffer as well. When returning to the container, the references to the syscall buffer are replaced with those to the original memory buffers. For buffers that are used to receive data from the OS, the data in the syscall buffer is copied to the original memory buffers as well.

Most system calls are interposed on by a single generic wrapper function in the CSM that uses a table of system call metadata to determine which arguments must be altered. System calls with more complex arguments, like those involving `iovec` structures are interposed on with more specific wrapper functions. On Linux, this interposing and altering of arguments works for most system calls with a few notable exceptions as discussed in Section 4.3.5.

When wrapping system calls, the CSM must ensure that system call errors are handled correctly. For simple error cases such as passing an invalid PID, the CSM only needs to pass the error

returned by the OS back to the caller. To handle more nuanced errors that can potentially affect the CSM, such as the program passing an invalid buffer address as a system call argument, the CSM must ensure all buffers passed via arguments are accessible to it when interposing on the system call. The CSM achieves this by first checking that buffer virtual addresses can be translated to valid physical addresses and, if the CSM needs to write to the buffer, it verifies that the memory does not need a CoW break before attempting to copy buffers, e.g. by using ARM's address translation instruction, `at`, which both translates and checks for write permission if requested. If this accessibility check fails, the buffer is either invalid, is CoW'ed, or not yet faulted into memory. To rule out the latter two conditions, the CSM records the buffer's virtual address and size in memory accessible to the OS, and control is returned to the OS. Prior to performing the actual system call, the OS checks if the CSM recorded any buffers and if so, breaks CoW if necessary and attempts to fault them in to ensure the CSM has access. The OS will then call the `enter_os` CSM call instead of attempting to perform the system call, allowing the CSM to again check if all buffers it needs are accessible and copy data to the task's syscall buffer. If a buffer is still not accessible to the CSM, the CSM will ignore it when wrapping the system call knowing that its inaccessibility is the result of a program error and will leave the invalid argument intact during interposition allowing the OS to detect that the requested memory is invalid when performing the system call and return the appropriate error.

As part of the copying of data from the OS to an enclaved container, BlackBox also does simple checks on system call return values to ensure they fall within predefined correct ranges. This has been shown to protect against many Iago attacks [166]. However, to keep its TCB simple and small, BlackBox only guarantees the correctness of system call semantics for memory management and inter-process communication (IPC), the latter discussed in Section 4.3.5. As a result, BlackBox protects against Iago attacks related to memory management and IPC, but is susceptible to some other Iago attacks. Augmenting BlackBox with a user-level runtime library in an enclaved container that guarantees the correctness of system call semantics could improve Iago attack protection, but at the cost of a larger TCB and potential additional limitations on system call

functionality.

4.3.5 Inter-process Communication

While BlackBox declassifies data to the OS passed in as system call arguments, it protects inter-process communication (IPC) among tasks running in the same enclaved container by encrypting the data passed into IPC-related system calls. This protects applications using IPC, which is transferred through and accessible to the OS. System calls that can create IPC-related file descriptors, such as `pipe`, and Unix Domain Sockets are interposed on and their returned file descriptors (FDs) recorded in per-process arrays marking them as related to IPC. When the CSM interposes on system calls that pass data through FDs, like `write` and `sendmsg`, it checks if the given FD is one related to IPC for that process. If it is, the CSM first uses authenticated encryption with a randomly generated symmetric key created during container initialization to encrypt the data before moving it into the task's syscall buffer. A record counter, incremented on each transaction, is included as additional authenticated data to prevent the host from replaying previous transactions. Similarly, data is decrypted and authenticated when interposing on system calls like `read` and `recvmsg` before copying it to the calling process's PPAS. With this mechanism, IPC communication is transparently encrypted and protected from the OS.

As mentioned in Section 4.3.4, to avoid trusting the OS's memory allocations, memory pages that are used by more than one process in a container are marked read-only in the container's PPAS unless the pages are known to belong to a shared memory mapping and are inherited during process creation. Shared memory regions created by a parent process through `mmap` with `MAP_SHARED` and faulted in prior to forking can be written to by both parent and child processes since the child's address space is validated after `fork`, as discussed in Section 4.3.3. However, for simplicity, BlackBox does not allow for writable IPC shared memory via XSI IPC methods such as `shmget` and `shm_open`, which are no longer widely-used. Modern applications instead favor thread-based approaches for performance or shared mappings between child worker processes via `mmap` compatible with BlackBox.

Futexes are used among threads and processes to synchronize access to shared regions of memory. As part of the design of `futex`, the OS is required to read the futex value, which is in the process's address space and included in the respective container's memory. This direct access to container memory is incompatible with BlackBox's memory isolation. To support `futex`, the OS makes a `futex_read` CSM call to obtain the value of a futex for container processes, rather than try and access the memory directly. The CSM ensures that only the futex address passed to `futex` can be read, and only if a `futex` call has been made.

Signals, used to notify processes of various events, present two issues for BlackBox. First, when delivering a signal to a process, a temporary stack for the signal handler is set up in the process's memory. With enclaved containers, this memory is not accessible to the OS. To remedy this, the OS is modified to setup this stack in a region of memory outside of the container's PPAS, which is then moved to the PPAS when the signal handler is executed and returned to the OS when the signal handler returns via `rt_sigreturn`. Second, the OS has to adjust the control flow of the process to execute the signal handler instead of returning to where it was previously executing. BlackBox cannot allow the OS to adjust the control flow of an enclaved process without validating it is doing so properly. To achieve this, as part of the CSM interposing on system calls, it tracks signal handler installation via system calls such as `rt_sigaction`. Upon handling a signal, the CSM ensures that the process will be correctly returning to a registered handler.

4.3.6 Container File System

Files within a container can only be accessed through an OS's I/O facilities making access to a container's files inherently untrustworthy without additional protection. A userspace encrypted file system could potentially be used to provide transparent protection of file I/O, but this would likely significantly increase the container's TCB. BlackBox relies on applications to use encryption to fully protect sensitive data files within a container, and provides a simple mechanism to allow the OS to load encrypted executable binaries for execution.

As discussed in Section 4.3.2, container images for BlackBox are pre-processed. For example,

ELF binaries, widely-used on Linux, have `.text`, `.data`, and `.rodata` sections that contain the executable code and data used by the code. These sections are combined into various segments when loaded into memory. In a BlackBox container image, the ELF headers are left unencrypted, but the `.text`, `.data`, and `.rodata` sections are encrypted then hashed, and their hash values are recorded along with the binaries. This enables BlackBox to validate the integrity and authenticity of executable binaries.

An ELF binary is executed by the OS as a result of a process calling `exec`, upon which the OS loads the binary by mapping its ELF headers into memory, reading the ELF headers to determine how to process the rest of the binary, then mapping the segments of the binary to memory. As discussed in Section 4.3.3, the OS is required to call `task_exec`, which passes the virtual addresses of the binary's loaded segments containing the `.text`, `.data`, and `.rodata` sections to the CSM. During this call, the CSM moves the process's pages, corresponding to the loaded binary, into the container's PPAS, validates that the hashes of the encrypted `.text`, `.data`, and `.rodata` sections match the hashes for the given binary from the BlackBox container image to confirm the authenticity and integrity of the loaded segments, then decrypts the sections in memory. The virtual to physical address mappings of these binary segments are recorded for later use. Upon returning from `task_exec`, the OS will begin running the task whose binary is now decrypted within protected container memory. If checking the hashes or decryption fails, the CSM will refuse to run the binary within an enclaved container, ensuring only trusted binaries can run within.

For dynamically linked binaries, in addition to the binary segments the OS maps during `exec`, the OS also maps the segments of the loader in the process's address space. These segments are verified in the same manner as the binary's segments. Dynamically linked binaries load and execute external libraries that BlackBox must validate are as expected and trusted. During the container image creation process, as with executable binaries, library binaries are also encrypted preventing their use without the CSM. These libraries are loaded and linked at runtime in userspace by a loader that is part of the trusted container image. To do this, the loader, running as part of a

process's address space, `mmap`s library segments into memory. The CSM intercepts these mmaps by interposing on FD-related system calls, such as `open`. If an FD is created for one of the libraries within a container, as recorded during container image creation, the CSM marks that FD as associated with the given library. If this FD is then used with `mmap`, the CSM intercepts it. Based on the size of the mmap request and the protection flags used, the CSM can infer which segment the loader is mapping. If it is a segment containing one of the encrypted sections, the CSM performs the same hashing, decryption, and memory map recording as it does with executable binaries.

4.4 Implementation

We have implemented a BlackBox prototype by repurposing existing hardware virtualization support available on modern architectures, including a higher privilege level, usually reserved for hypervisors, and nested page tables (NPTs). NPTs, also known as Arm's Stage 2 page tables and Intel's Extended Page Tables (EPT), is a hardware-assisted virtualization technology that introduces an additional level of virtual address translation [4]. When NPTs are used by hypervisors, the guest OS in a VM manages its own page table to translate a virtual address to what the VM perceives as its physical address, known as a guest physical address, but then the hypervisor manages an NPT to translate the guest physical address to an actual physical address on the host. Hypervisors can thereby use NPTs to control what physical memory is available to each VM.

BlackBox uses hardware virtualization support to run the CSM in lieu of a hypervisor to support PPASes. The CSM runs at the higher hypervisor privilege level, so that it is strictly more privileged than the OS and is able to control NPTs. The CSM introduces an NPT for each container and the OS, such that a container's PPAS is only mapped to its own NPT, isolating the physical memory of each container from the OS and each other. The CSM switches a CPU from one PPAS to another by simply updating its NPT base register to point to the respective container's NPT. Similarly, the CSM uses NPTs to protect its own memory from the OS and containers by simply not mapping its own memory into the NPTs. The memory for the NPTs is part of the CSM's protected memory and is itself not mapped into any NPTs so that only the CSM can update the NPTs. When the CSM

runs, NPTs are disabled, so it has full access to physical memory.

Specifically, BlackBox uses Arm hardware virtualization extensions (VE) [3, 99, 98, 167]. The CSM runs in Arm's hypervisor (EL2) mode, which is strictly more privileged than user (EL0) and kernel (EL1) modes. EL2 has its own execution context defined by register and control state, and switching the execution context of EL0 and EL1 are done in software. The CSM configures Stage 2 page tables in EL2, and the System Memory Management Unit (SMMU), Arm's IOMMU. The Linux kernel runs in EL1 and has no access to EL2 registers, so it cannot compromise the CSM. CSM calls are made using Arm's `hvc` instruction from EL1.

Before and after every transition to the OS, BlackBox traps to the CSM, which in turn switches between container and OS NPTs. One might think that imposing two context switches to the CSM to swap NPTs for every one call to the OS would be prohibitively expensive, but we show in Section 4.5 that this can be done on Arm without much overhead. The flexibility that Arm EL2 provides of allowing software to determine how execution context is switched between hypervisor and other modes turns out to be particularly advantageous for implementing the CSM because it does not lock its implementation into using heavyweight hardware virtualization mechanisms to save and restore hypervisor execution context that are not required for the CSM.

Trapping to the CSM before and after every transition to the OS requires that the CSM interpose on all system calls, interrupts, and exceptions. Hypervisors traditionally accomplish similar functionality by trapping interrupts and exceptions to itself, then injecting virtual interrupts and exceptions to a VM. BlackBox avoids the additional complexity of virtualizing interrupts and exceptions by taking a different approach. The CSM configures hardware so system calls, interrupts, and exceptions trap to the OS and modifies the OS's exception vector table for handling these events so that `enter_os` and `exit_os` CSM calls are always made before and after the actual OS event handler. To guarantee these handlers are installed and not modified by the OS at a later time, BlackBox requires the OS to make a `protect_vectors` CSM call with the address of the text section of the vector table during system initialization, before any container may be enclaved. The CSM then prevents the OS from tampering with the modified vector table by marking

its backing physical memory read only in the OS's NPT. Similarly, the vDSO region of memory is marked read only to prevent malicious tampering of the region.

Figure 4.3 depicts the steps involved in interposing on transitions between the containers and OS when repurposing virtualization hardware. While running in a container, an exception occurs transferring control to the protected OS exception vector table (step 1). All entry points in the exception vector table invoke the `enter_os` CSM call (step 2). During this, the CSM switches to the OS's NPT (step 3). The OS will therefore not be able to access private physical memory mapped into container NPTs. For system call exceptions, system call arguments are copied to an OS accessible syscall buffer (step 4). Control is transferred back to the OS (step 5) to perform the required exception handling. When the OS has finished handling the exception, the `exit_os` CSM call is made as part of the return path of the exception vectors when returning to userspace (step 6). For system call exceptions, OS updated arguments are copied back to the original buffer (step 7). On `exit_os`, the CSM verifies the exception return address to ensure the call is from the trusted exception vectors, which the OS cannot change, rejecting any that are not. The CSM then checks if the running task belongs to an enclaved container, in which case the CSM switches to the respective container's NPT so the task can access its PPAS memory state (step 8). Control is restored to the container by returning from `exit_os` (step 9) and back to userspace (step 10). If `exit_os` is not called, the CSM will not switch the CPU to use the container's PPAS, so its state will remain inaccessible on that CPU.

BlackBox protects a container's memory by using separate NPTs for the OS and each container, but still relies on the OS to perform all complex memory management functions, such as allocation and reclamation, to minimize the complexity and size of the CSM. This is straightforward because unlike hypervisors which virtualize physical memory using NPTs, the CSM merely uses NPTs for access control so that the identity mapping is used for all NPTs including the OS's NPT. The OS's view of memory is effectively the same as the actual physical memory for any physical memory mapped into the OS's NPT. Except for the CSM's physical memory, all physical memory is initially assigned to the OS and mapped to its NPT. When the OS allocates physical memory

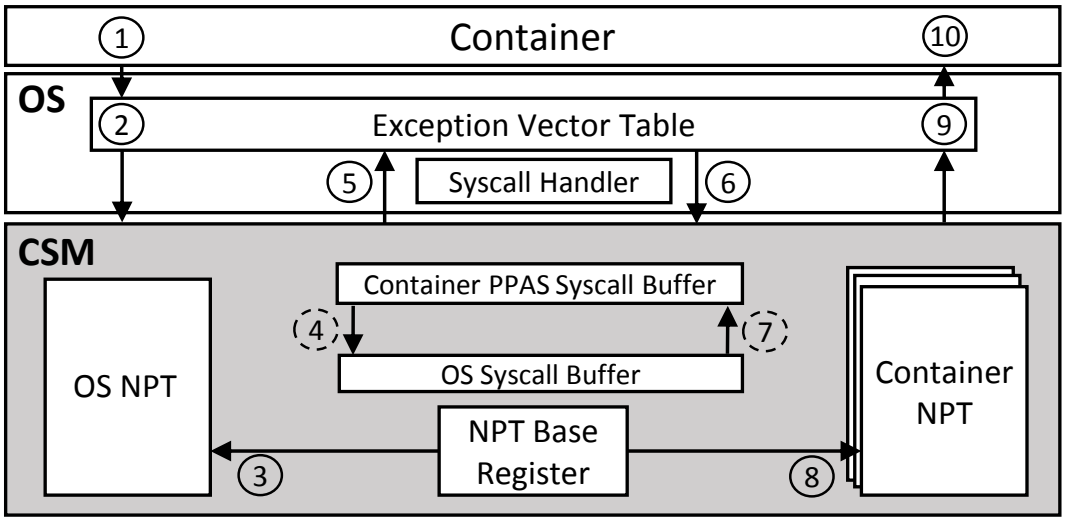

Figure 4.3: System Call from Enclaved Container.

to processes in containers, the CSM can just unmap the physical memory from the OS's NPT and map it to the respective container's NPT at the same address. The CSM does not need its own complex allocation functionality. The CSM checks the OS's NPT to make sure that the OS has the right to allocate a given page of memory. For example, should the OS attempt to allocate a physical page belonging to the CSM, the CSM will reject the allocation and not update the OS's or container's NPT. The CSM also checks that any page allocation proposed by the OS for a container is not mapped into the IOMMU page tables and will therefore not be subject to DMA attacks, as discussed in Section 4.3.1.

Note that the OS is oblivious to the fact that its allocation decisions for process page tables, Arm's Stage 1 page tables, are also used for Stage 2 page tables. Furthermore, since Arm hardware first checks Stage 1 page tables before Stage 2 page tables, page faults due to the need to allocate physical memory to a process all appear as Stage 1 page faults, which are handled in the normal way by the OS's page fault handler. Since the CSM maps the physical memory to the respective Stage 1 and Stage 2 page table entries at the same time, there are no Stage 2 page faults for memory allocation.

As discussed in Section 4.3.4, BlackBox requires that process page tables cannot be directly modified by the OS. At the same time, commodity OSes like Linux perform many operations that

involve walking and accessing process page tables. To minimize OS modifications required to use enclaved containers, BlackBox makes the process page tables readable but not writable by the OS by marking the corresponding entries in the OS's NPT read only. All existing OS code that walks and reads process page tables can continue to function without modification, and only limited changes are required to the OS to use CSM calls for any updates to process page tables. A process's page tables are also mapped to its respective container's NPT, so they can be accessed by MMU hardware for virtual address translation while executing the process. BlackBox also maps tasks' syscall buffers, used for passing system call arguments to and from the OS, to their Stage 1 page tables. This allows OS functions designed to copy data to and from buffers in the calling process's address space to function correctly without modification. The tasks' syscall buffers themselves are only mapped to the OS's NPT, not the container's NPT, as they are shared directly only by the CSM and OS.

To optimize TLB usage, physically contiguous memory can be mapped to an NPT in blocks larger than the default 4 KB page size. The BlackBox implementation supports transparent 2 MB stage 2 block mappings by first fully populating the last-level stage 2 page table with 4 KB mappings, then folding all 512 entries into a single entry. BlackBox checks that all 512 entries are contiguous in physical memory and that the first entry is aligned to a 2 MB boundary. BlackBox will unfold a block mapping if one of the original 512 entries is unmapped, such that all 512 entries are no longer contiguous in physical memory. Similarly, BlackBox will unfold a block mapping if there is a need to change the attributes of one of the original 512 entries, such as marking it read only while other entries remain writable. This approach is advantageous over just supporting huge pages allocated by the OS because it improves TLB usage even when the OS does not use huge pages.

BlackBox's implementation is relatively small. The implementation is less than 10K lines of code (LOC), most of which is the 5K LOC for the implementation of Ed25519, ChaCha20, and Poly1305 from the verified HACL* crypto library [168]. Other than HACL*, BlackBox consisted of 4.9K LOC, all in C except for 0.4K LOC in Arm assembly. Table 4.2 shows a breakdown by

feature. 0.3K LOC was for verifying the CSM was correctly booted and initialized. 1K LOC was for enclave management, including enclave creation and handling enclave metadata 0.1K LOC was for switching between enclaves and the OS. 0.2K LOC was for protecting data in CPU registers. 1K was for system call interposition, including marshaling of arguments. The table used for determining how to marshal system calls and check return values is dynamically generated as a single line of C code at compile time. 2.3K LOC was for memory protection, including NPT management of PPASes, Iago and DMA protection, and handling and validating page table update requests. BlackBox's CSM TCB implementation complexity is similar to other recently verified concurrent systems [169, 170, 171, 172], suggesting that it is small enough that it can be formally verified. Beyond the CSM itself, only 0.5K LOC were modified or added to the Linux kernel to support BlackBox.

Table 4.2 also compares the code complexity of BlackBox versus the Linux kernel and KVM hypervisor. This is a conservative comparison as the LOC for Linux and KVM only include code compiled into the actual binaries for one specific Arm server used for the evaluation in Section 4.5. Even with this conservative comparison, BlackBox is orders of magnitude less code, in part because its functionality is largely orthogonal to both OSes and hypervisors, which have much more complex functionality requirements.

4.5 Experimental Results

We quantify the performance of BlackBox compared to widely-used Linux containers, and demonstrate BlackBox's ability to protect container confidentiality and integrity. Experiments were run using both Arm multiprocessor embedded system and server hardware with VE support, specifically (1) a Raspberry Pi 4 Model B with a 4-core Cortex-A72 64-bit 1.5 GHz Broadcom BCM2711 SoC, 8 GB RAM, a 250 GB Samsung 860 EVO SSD connected via USB3.0, and Gigabit Ethernet, running Raspberry Pi OS Buster (2020-08-20 Debian), and (2) an AMD Seattle Rev.B0 server with an 8-core Cortex-A57 64-bit ARMv8-A 2 GHz AMD Opteron A1100 SoC, 16 GB of RAM, a 512 GB SATA3 HDD, and an AMD XGBE 10 GbE NIC, running Ubuntu

Feature	BlackBox	Linux	KVM
Bootstrapping	0	8.5K	8.5K
Device Support	0	425K	425K
Filesystem Support	0	163K	163K
Process Management	0	110K	110K
Memory Management	0	60.7K	60.7K
CPU Scheduling	0	29.3K	29.3K
Networking	0	190K	0
Sound	0	89.3K	0
Process Security	0	64.7K	0
Device Virtualization	0	0	30.1K
CPU Virtualization	0	0	3.5K
VM Switch	0	0	1.2K
Cryptography	5K	19K	19K
Boot Verification	0.3K	0	0
Enclave Management	1K	0	0
Enclave Switch	0.1K	0	0
CPU Protection	0.2K	0	0
Syscall Interposition	1K	0	0
NPT Management	1K	0	2.8K
Memory Mapping Protection	0.5K	0	0
DMA Protection	0.8K	0	9K
Total	9.9K	1.2M	862K

Table 4.2: LOC for BlackBox, Linux, and KVM.

16.04. For client-server experiments, the clients ran on a Lenovo ThinkPad P52 with a quad-core Intel i7-8750H 64-bit 4.1 GHz CPU, 32 GB RAM, and a 1 TB PCIe SSD, running Linux Mint 20, connected to the Arm hardware via Gigabit Ethernet through an ASUS RT-N16. All machines used Linux kernel 5.4 LTS and for running in containers, the Docker 20.10.6 container runtime.

We ran the microbenchmarks and application workloads listed in Table 4.3 using the following five system configurations: (1) natively on the host without containers to provide a baseline measure of performance, (2) Docker with unmodified Linux containers (Docker), (3) BlackBox running Docker with traditional Linux containers, without the security guarantees of being enclaved (BlackBox NS, for Non-Secure), (4) BlackBox running Docker with enclaved Linux containers without encrypted IPC (BlackBox NE, for no encryption), and (5) BlackBox running Docker with enclaved Linux containers (BlackBox Enclaved). Three BlackBox configurations were used to quantify the cost of different protection mechanisms. BlackBox NS provides the same security

Name	Description
Lmbench	`lmbench` v3.0-a9 [173] latency microbenchmarks.
Hackbench	`hackbench` [174] using Unix domain sockets and 100 process groups running in 500 loops.
Apache	`Apache` v2.4.46 server handling 100 concurrent requests from remote `ApacheBench` [175] v2.3 client, serving the 12 KB default Debian `index.html`.
HAProxy	`HAProxy` v1.8.19 server proxying 100 concurrent requests from remote `ApacheBench` [175] v2.3 client to remote `Apache` v2.4.29, serving the 82 KB `index.html` of the GCC 13.0.0 manual.
Kernbench	Compilation of the Linux 5.4 kernel using `allnoconfig` for Arm with GCC 8.3.0.
Memcached	`memcached` v1.6.9 using the `memtier` [176] benchmark v1.3.0 with default parameters.
MySQL	`MariaDB` v10.3.27, a `MySQL` fork, handling requests from remote `YCSB` [177] v0.17.0 client running workload A with 200 parallel transactions, recordcount=500K, and opcount=100K.
Netperf	`netperf` v2.6.0 [178] running `netserver` on the server and the client with default parameters in three modes: TCP_STREAM (receive throughput), TCP_MAERTS (send throughput), and TCP_RR (latency).
Nginx	`Nginx` v1.18.0 server handling 100 concurrent requests from remote `ApacheBench` [175] v2.3 client, serving the 12 KB default Debian `index.html`.

Table 4.3: Microbenchmarks and Application Workloads.

as Docker, the only difference being that BlackBox NS runs the containers on BlackBox with the OS's NPT enabled, to quantify NPT overhead. BlackBox NE provides stronger security by enclaving the container but without enabling IPC encryption, thereby quantifying BlackBox overhead without IPC encryption. BlackBox Enclaved is the same as BlackBox NE but with IPC encryption enabled. When using BlackBox, its DMA protection is not available on the Raspberry Pi 4 because it has no SMMU. Docker's default `seccomp` policy is enabled for all configurations. Versions of `libseccomp` prior to v2.5 had a significant performance issue on policies like Docker's default [179]. The Docker version we use incorporates this performance fix.

4.5.1 Performance Measurements

Figure 4.4 shows performance measurements for each microbenchmark for each container configuration normalized to native execution; lower numbers are better. Solid bars indicate results run on the Raspberry Pi and the overlaid outlined bars indicate results run on the AMD Seattle Arm server. BlackBox has the highest overhead relative to native execution on the null system call mea-

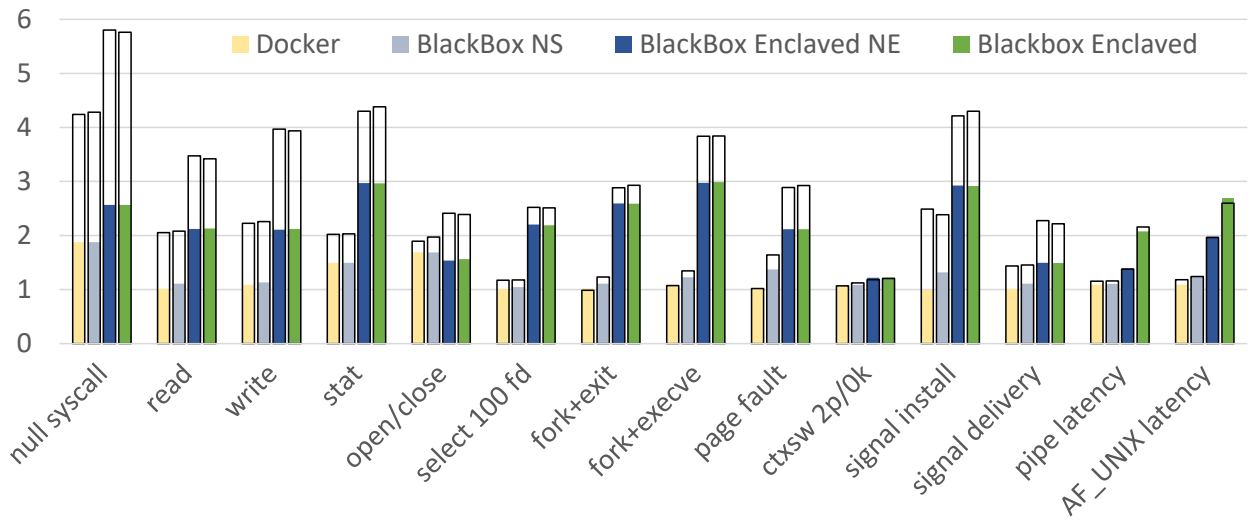

Figure 4.4: Container Performance for Microbenchmarks.

surement, but most of the overhead is from Docker, due to its use of `seccomp` to configure and limit the system calls available in a container to reduce the available attack surface area. Although `seccomp` is used for all system calls, its overhead is most apparent for the null system call as its base cost is the lowest since it does no work. In contrast, the overhead due to BlackBox, from the two CSM calls that BlackBox makes on every system call, is small relative to `seccomp`. Although CSM calls require switching to and from Arm's EL2 mode, it requires no more than EL2's system register state to execute, eliminating the need to save and restore system registers when switching between EL1 and EL2; only general-purpose registers need to be saved and restored. Taking advantage of Arm's architectural features makes CSM calls relatively inexpensive, enabling fine-grained container protection without significant overhead from system call interposition. The key aspect of Arm's design that is crucial for the CSM is that software determines what state needs to be saved and restored. Running the CSM in the equivalent x86 hypervisor root mode would be much more expensive as it provides a hardware instruction that must be used to context switch to root mode that requires saving and restoring the entire CPU system state [99]. The x86 mechanism works well for hypervisors since they already require this operation, but poorly for the CSM which makes minimal use of CPU system state, and therefore does not need the expensive save

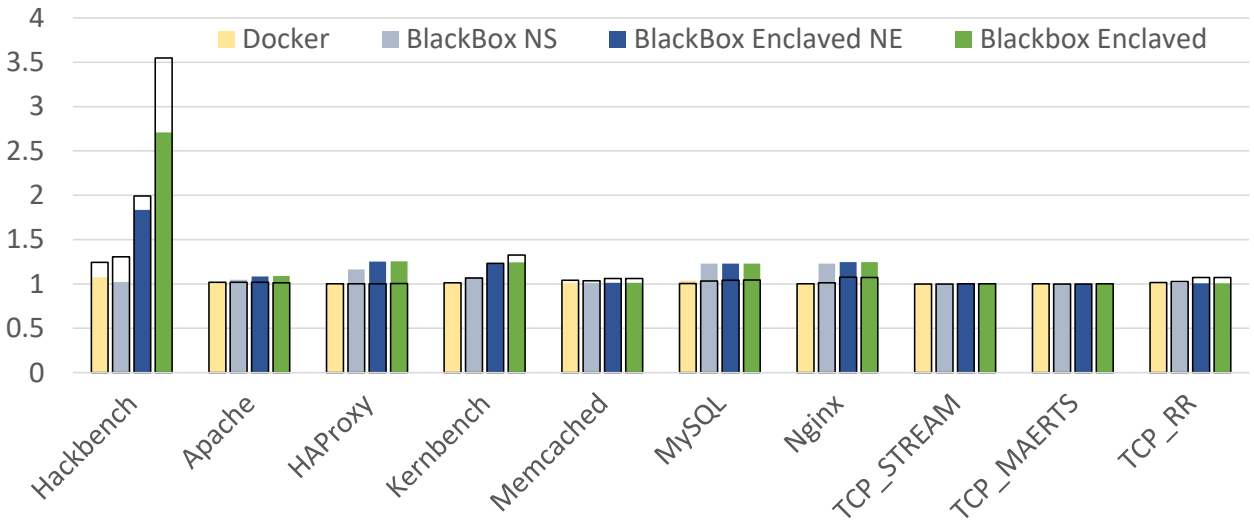

Figure 4.5: Container Performance for Application Workloads.

and restore.

For the `read`, `write`, `stat`, `open/close`, and `select` system call measurements, BlackBox Enclaved is less than two times the cost of Docker. The overhead for the enclaved configurations is due to the need to copy system call arguments back and forth between the container PPAS and OS, since enclaved container memory is not accessible to the OS. `open` additionally incurs overhead as part of checking the path being opened to identify FDs associated with shared libraries as part of BlackBox's binary decryption mechanism. For all system calls, the overhead on the AMD server, as indicated by the outlined bars, exceeds that of the Raspberry Pi's. In most cases, this is due to the server hardware performing the CPU bound system call operations more quickly than the Raspberry Pi while their memory performance remains similar, resulting in the similar costs for BlackBox's system call argument copying having relatively higher overhead.

`fork` and `exec` measurements show the highest overhead for BlackBox Enclaved versus Docker, less than three times the cost of Docker. This is due to validating that the new process's address space matches its parent's on `fork`, and additionally validating the address space against the new binary's mappings on `exec`. Although the binary must be decrypted for `exec` measurements, it is only decrypted once and all subsequent iterations just confirm the mappings match the

first's, thereby amortizing the cost of the initial decryption.

Page fault measurements show the one microbenchmark for which there is noticeable overhead for BlackBox NS versus Docker. This is due to the added cost of using NPTs for the BlackBox NS configuration. This overhead then increases for enclaved containers due to needing to verify the fault resides within a known address mapping to protect the container from potential Iago attacks from the OS. Although a page fault results in several context switches to the CSM, the context switches themselves are not a significant cost because they are relatively inexpensive on Arm.

Protecting container IPC communication through encryption imposes little cost for most workloads, but this overhead is noticeable for pipe, UNIX domain sockets (AF_UNIX), and `hackbench` measurements. These benchmarks represent worst-case overheads for IPC encryption because they all use IPC to read and write a single byte to signal other processes. When encrypting, this single byte is padded and written along with authentication data, significantly increasing the relative write size and affecting read/write latency measurements. In contrast, the context switch microbenchmark, in which a parent process spawns two child processes that communicate between each other with pipes, has almost no overhead. In this case, 4 byte reads and writes are used so the extra data that encryption adds, and therefore the time to complete the calls, is relatively less, and context switching and rescheduling dominates IPC encryption costs. The signaling microbenchmarks do not involve any encryption. BlackBox Enclaved overhead for signal installation is due to copying the `sigaction` struct in and out, and for signal delivery is due to verifying the control flow.

Figure 4.5 shows performance measurements for each application workload for each container configuration normalized to native execution; lower numbers are better. Apache, HAProxy, Kernbench, memcached, MySQL, and Nginx measurements show that BlackBox overhead is much less on realistic application workloads than microbenchmarks. In most cases, BlackBox Enclaved overhead versus native execution is less than 15% on both the Raspberry Pi and AMD server, demonstrating modest overhead across both Arm embedded and server hardware. As indicated by the BlackBox NS measurements, NPT usage is a source of overhead, though more so on the

Raspberry Pi than the AMD server. Apache, HAProxy, and Nginx workloads measure latency in addition to throughput. In terms of latency, the overhead for these workloads for BlackBox Enclaved versus native execution is less than 15% on both the Raspberry Pi and AMD server. Furthermore, Netperf measurements show that BlackBox provides fast networking performance as it involves no I/O virtualization, in contrast to using VMs. Applications are able to make full use of the host's networking capabilities. Although applications are expected to encrypt their network I/O to protect their data, we did not encrypt network connections for these measurements to avoid encryption costs obscuring BlackBox's overhead.

Figure 4.6 quantifies the CPU utilization when running the application workloads, as a measure of computational overhead. Solid bars indicate results run on the Raspberry Pi and the overlaid outlined bars indicate results run on the AMD server. CPU utilization is generally lower on the AMD server than the Raspberry Pi, since the AMD server is more powerful with more CPUs. On the Raspberry Pi, the difference in CPU utilization between BlackBox Enclaved and native execution is less than 15% across all workloads, and less than 5% for all workloads except Apache and Memcached. On the AMD server, the difference in CPU utilization between BlackBox Enclaved and native execution is less than 15% across all workloads, except Apache. Apache CPU utilization for BlackBox Enclaved is high because at higher throughput rates, the cost of extra copying to use syscall buffers, as discussed in Section 4.3.4, becomes dominant. The buffers are used to send data from a container's PPAS to the OS to perform network I/O. Other than Apache, the difference in CPU utilization between BlackBox Enclaved and native execution is quite modest across both Arm embedded and server hardware.

4.5.2 System Call Coverage

We evaluated the completeness of Linux system call support in the current BlackBox prototype implementation by running the Linux Test Project (LTP) [180] version 20210524 system call test suite. LTP consists of 1344 test cases designed to test for correct functionality across the entire Linux system call interface. We compared system call support results for running LTP in an en-

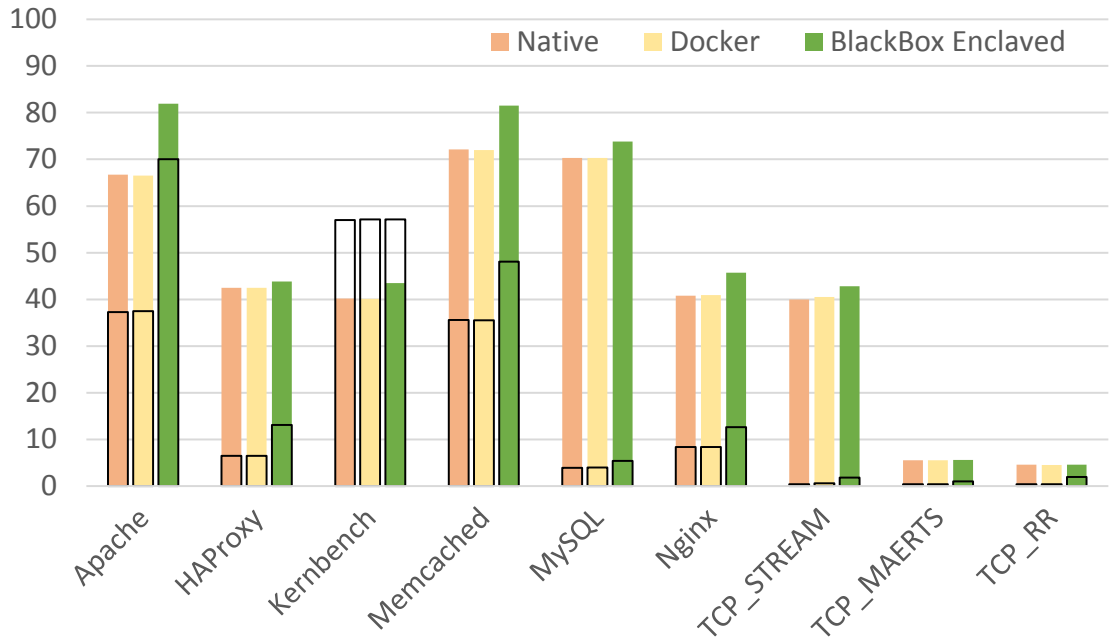

Figure 4.6: CPU Utilization for Application Workloads.

claved container on BlackBox versus running it natively, in both cases using the Raspberry Pi. When running LTP natively, 1149 test cases pass and 195 fail. These failures are expected and are a combination of missing dependencies and unsupported features of the kernel and architecture used. For example, test cases for the 16-bit version of `fchown` are not supported on the platform. When running LTP using BlackBox, 1012 test cases pass and 332 fail, demonstrating support for almost 90% of test cases that passed when run natively. The additional 137 failed tests are due to the current prototype not yet supporting lesser used system calls like `process_vm_readv`.

4.5.3 Evaluation of Practical Attacks

We evaluated BlackBox's effectiveness against a compromised OS by analyzing CVEs related to the Linux kernel and various Linux container engines such as Docker. We considered 23 CVEs which could result in privilege escalation, code execution, and memory corruption in Linux capable of compromising the integrity and confidentiality of container data; we did not consider denial of service attacks, as BlackBox does not guarantee availability. Specifically, privilege escalation occurs if the exploit enables the attacker to gain root access or kernel privilege level, and code

Bug	Description	Containers	BlackBox
CVE-2009-3234	Kernel buffer overflow enabling return-to-user attack.	No	Yes
CVE-2010-2959	Function pointer overwrite due to integer overflow.	No	Yes
CVE-2010-4258	Kernel memory overwrite due to improper handling of get_fs value.	No	Yes
CVE-2013-6441	Improper permissions when mounting /sbin/init.	No	Yes
CVE-2014-6407	Symbolic and hardlink issues during docker pull.	No	Yes
CVE-2014-9357	Mishandling untrusted archive extraction.	No	Yes
CVE-2015-1335	Directory traversal flaw in lxc-start.	No	Yes
CVE-2015-3627	Unchecked file descriptor opened prior to chroot.	No	Yes
CVE-2015-3629	Unchecked symlink in image when respawning container.	No	Yes
CVE-2015-3630	Weak permissions on /proc filesystem.	No	Yes
CVE-2016-1576	Improperly restricted mount namespace.	No	Yes
CVE-2016-5195	Race condition in Linux's handling of copy-on-write breakage.	No	Yes
CVE-2016-7117	Use after free in __sys_recvmmsg.	No	Yes
CVE-2016-9962	Improperly flushed file descriptors.	No	Yes
CVE-2017-7308	Improper validation of data size in Linux's packet_set_ring().	No	Yes
CVE-2017-1000112	Exploitable memory corruption due to UFO to non-UFO path switch.	No	Yes
CVE-2018-15664	TOCTOU vulnerability in symbolic link checking.	No	Yes
CVE-2018-18955	Mishandled nested user namespaces in Linux's map_write().	No	Yes
CVE-2019-5736	/proc/self/exe file descriptor mishandling	No	Yes
CVE-2019-10144	Container processes not isolated during 'rkt enter' command.	No	Yes
CVE-2019-11247	Improper access control to cluster-scoped custom resources.	No	Yes
CVE-2019-14271	Container contents loaded while privileged during container copy.	No	Yes
CVE-2020-14386	Kernel memory corruption due to arithmetic issue in tpacket_rcv().	No	Yes

Table 4.4: CVEs Used for Evaluation of Practical Attacks.

execution occurs if the exploit enables executing arbitrary code at the same privilege as the software with the bug.

Table 4.4 lists the CVEs considered. We considered both malicious containers and unprivileged host users who exploit bugs in the kernel and container engines to elevate privileges and compromise container data. In general, these CVEs exploit flaws in container runtime systems and the kernel that enable an attacker to obtain kernel-level or root-level access. Ordinarily, this level of access compromises all container data and integrity on the system. Linux and the relevant container engine do not fully protect against any of these compromises. In contrast, BlackBox protects against all of them.

4.6 Related Work

Various approaches have been explored to securing applications from untrusted OSes. Hardware-based trusted execution environments (TEEs) such as ARM TrustZone [138] and Intel Software

Guard Extensions (SGX) [137] can protect application memory from higher privileged software, but require applications to be written or rewritten specifically for this purpose and may impose other functionality restrictions.

Some systems have built on TEEs. Haven [139] aims to enclave Windows applications by porting a Windows library OS to run inside SGX, avoiding Iago attacks by trusting the library OS at the cost of a significant TCB. Other systems also propose running library OSes enclaved by SGX [181, 182, 183]. CubicleOS[184] is a library OS designed to be runnable within containers that makes use of Intel MPK hardware extensions to isolate apps. Scone [140] uses SGX to enclave Linux containers, requiring its own custom threading model and a modified C library within SGX to provide system call support and shielded I/O interfaces for interacting with the OS. TZ-Container [185] leverages a shield layer and a container manager inside TrustZone to protect containers, but relies on the OS not modifying the memory mappings used to protect containers by scanning the OS image to ensure it does not contain instructions capable of updating page tables. TrustShadow [186] introduces a runtime system within TrustZone so that a limited number of security-critical legacy apps operate on TrustZone memory isolated from the OS. Unlike these approaches, BlackBox does not rely on TrustZone or SGX and does not rely on a library OS or other significant runtime system running inside an enclaved execution environment, avoiding increasing TCB complexity. Unlike Haven, its small TCB comes with potentially greater susceptibility to Iago attacks by allowing applications to use the system call interface of the untrusted OS.

Commodity hypervisors have been modified to secure applications from an untrusted OS by restricting a guest OS in a VM to an encrypted view of application memory [141, 147, 142, 143, 145, 187, 188]. For example, InkTag [142] uses two NPTs as part of its isolation mechanism, one for the OS and the other for all applications, separating the plaintext memory of isolated applications from encrypted memory, but relying on paravirtualized page table updates to isolate applications from each other. Appshield [144] uses virtualization techniques to protect and isolate critical applications against OS-level malware attacks. Appshield's memory protection model requirements are not compatible with Linux's copy-on-write semantics and its limited system call interface is

insufficient to support significant workloads. In contrast, BlackBox does not rely on a hypervisor or traditional memory virtualization, but instead introduces a new concept of protected physical address spaces implemented as part of a container security monitor, enabling it to have a much smaller TCB.

Various approaches reduce the hypervisor's TCB. Microhypervisors [189, 146, 190] build new hypervisors from scratch with smaller TCBs, but at the cost of a significantly reduced feature set. BlackBox's approach allows for a small TCB while still maintaining a significant feature set and the full hardware support available in a commodity OS. SeKVM [191, 169, 170, 171] retrofits KVM with a small verified TCB to provide VM data confidentiality and integrity. In contrast, BlackBox provides container-level isolation and does not require a hypervisor, introducing a new concept, the CSM, that avoids the cost and complexity of hypervisor-based virtualization.

X-Containers [192] targets securely isolating containers in the cloud. Its containers include an entire library OS based on Linux and run on top of a Xen hypervisor, providing a model more akin to nested virtualization. Unlike BlackBox, X-Containers have a large TCB from requiring both large library OSes and a commodity hypervisor.

Other approaches have looked at ways to harden traditional containers. gVisor [193] runs a limited userspace kernel within a container and beneath applications. System calls are intercepted to further isolate applications from the host OS through reduced interactions and potential attack surfaces. gVisor's increased isolation comes at the cost of a increased TCB size in the container. Distroless images [194] aim to limit the contents of a container to precisely what is necessary for the target app to run, reducing what must be trusted and maintained within a container. Linux Container Hardening [195] aims to improve the security of Linux containers through improving the kernel subsystems and primitives used by containers to be more secure. These approaches are complementary to BlackBox, and although they improve container security, unlike BlackBox, they all must still trust the OS and its large codebase.

4.7 Summary

BlackBox is a new container architecture providing fine-grain protection of application data confidentiality and integrity without trusting the OS. BlackBox achieves this by introducing a container security monitor, a new software component that creates protected physical address spaces for containers. The monitor enforces protected address spaces to isolate container memory and CPU state from the OS and other containers. It facilitates the use of OS facilities via system calls by passing required data between protected address spaces and the OS, implicitly declassifying such data. This narrow purpose keeps it small and simple. Unlike a hypervisor, the monitor performs no virtualization or resource management. Instead, it relies on the OS to provide complex functionality required to manage hardware resources, including CPU scheduling, memory management, file systems, and device management. We have implemented BlackBox by repurposing Arm hardware virtualization support. Our results demonstrate that BlackBox supports existing unmodified containerized application workloads with modest overhead while maintaining a trusted computing base orders of magnitude less than an OS or commodity hypervisor.

Chapter 5: Conclusions and Future Work

5.1 Conclusions

Through three key systems, my research has demonstrated that container architectures can be created to enable new functionality and greater security across a range of platforms. These architectures leverage the advantages containers have over VMs to achieve their goals, including modest hardware requirements, a low resource footprint, and tighter coupling with the virtualization layer enabling these architectures to utilize existing code to minimize implementation effort.

First, we presented Flux, a system supporting multi-surface apps through app migration. Flux introduces two novel mechanisms. *Selective Record/Adaptive Replay* to record just those device-agnostic app calls that lead to the generation of app-specific device-dependent state in services and replay them on the target. Selective Record/Adaptive Replay leverages small compile-time decorator modifications to system service IPC definitions enabling Flux to interpose on app calls to system services and only record those that modify app-specific device state while automatically discarding stale interactions. *Checkpoint/Restore in Android (CRIA)* to transition an app into a state in which what little device-specific information the app contains can be safely discarded before checkpointing and restoring the app within a containerized environment. CRIA checkpoints and restores critical user- and OS-level state of a running app along with app-specific state held within Android specific device drivers. We implemented and evaluated a working prototype of Flux on Android. Flux is able to migrate unmodified popular apps in a modest amount of time with minimal runtime overhead during app execution.

Second, we presented AnDrone, a drone-as-a-service solution making drones accessible in the cloud. AnDrone provides users with cloud-configurable lightweight *virtual drones* supported by a device container design. Virtual drones are containerized instances of Android Things that

appear to users as if it is the only software running on and controlling a drone. Various apps and services of interest to a user for controlling the drone can be added to these virtual drones. To support the device container design, AnDrone transparently decouples apps from low-level device implementation and interfaces through its *device container* architecture. The device container encapsulates all physical drone devices in a separate isolated execution environment and controls access to devices to isolate virtual drones from one another and preserve drone hardware safety. Experimental results show AnDrone is scalable, incurs little to no overhead, and can reliably isolate a flight controller.

Finally, we presented BlackBox, a container architecture providing app data confidentiality and integrity on an untrusted OS. BlackBox introduces a new *container security monitor (CSM)* mechanism to create separate and independent physical address spaces for each container. The CSM leverages existing hardware virtualization features to run at a higher privilege level than the OS to provide containers with protected physical address spaces (PPASes). The CSM interposes on all interactions between protected containers and the OS, enabling it to adjust which PPAS is active when running. We implemented BlackBox on Arm hardware, demonstrated modest overhead, and showed that BlackBox can support widely-used Linux containers with only minor modifications to the Linux kernel while inheriting support for a broad range of Arm hardware from the OS.

5.2 Future Work

My research demonstrated the feasibility of new container architectures that enable additional functionality and provide greater security across a range of platforms. In building these systems, it became apparent that there exist opportunities to extend and apply some of the novel concepts of these architectures to other areas of research.

AnDrone's virtual drones potentially allow broader use cases and offer greater accessibility to drones. An important issue that needs to be further researched though is the physical security of the drone. AnDrone offers geofences for flight control, but this is only a limited solution to the overall safety of the physical drone. Self-driving technology used in cars, such as automatic braking

in the case of an imminent impact, may be useful to offer a more complete solution. However, these systems are insufficient for drones. Unlike cars, drones operate in three-dimensions, so an additional vertical dimension of operation must be taken into account. Additionally, drones are much more agile and maneuverable than cars and so significantly faster methods of detecting unsafe conditions and responding to them are required. Implementing a comprehensive physical safety solution for AnDrone will need to address these challenges.

AnDrone supports multiple virtual drones by running each virtual drone within a container. Therefore, it suffers from the same security shortcomings as compared to VMs that BlackBox was created to remedy. AnDrone would benefit from the additional security offered to containers by BlackBox if BlackBox were capable of supporting entire virtual drones. A complication to this is that virtual drones are full Android Things OS instances. They run a significant number of processes and require the use of Android-specific drivers, like Binder. For the support and use of Android-specific drivers, at a minimum, additional device-specific `ioctl` system call wrappers need to be created within the CSM. However, multiple Android drivers make extensive use of shared memory, including Binder. Given BlackBox's restrictions on shared memory, a key challenge will be how to support Android's more extensive shared memory usage while maintaining BlackBox's security and performance for virtual drones.

BlackBox protects apps in containers from an untrusted OS, minimizing the TCB to just that of the CSM and the contents of the container. However, much of the code in a container is typically large third-party libraries that may contain vulnerabilities putting the app within the protected container at risk if they are exploited. To further reduce the TCB and protect against these third-party library vulnerabilities, the concept of BlackBox protecting an enclave from an untrusted OS can be made finer-grained such that the CSM protects apps from third-party libraries in a manner similar to how it protects against the OS. A shim library for each third-party library can be created for an app to link against. The shim library in turn then links against the actual libraries. When the app makes a library call, the shim acts similarly to the protected OS exception vector table and the system call wrappers within the CSM. However, unlike system calls which have a

fixed calling convention for a given architecture, the calling convention used within userspace can vary. A key challenge to protecting against untrusted libraries is how to support library-specific calling conventions in a general way. Additionally, userspace has increasingly become reliant on heavy usage of asynchronous APIs. Supporting these APIs while remaining performant, general, and reusable so as to not require burdensome implementation effort for each API is another key research challenge that will need to be addressed to protect against untrusted libraries.

While BlackBox focuses on protecting CPU and memory state, applications are increasingly running code and manipulating data using GPUs. An important research challenge to explore is how to extend the security guarantees of container data confidentiality and integrity to applications using GPUs for general-purpose computing. TEEs for protecting GPU data have been proposed [196, 197], but currently remain research only and have not been built. Even if such TEEs existed, there are still side-channel concerns that can enable an attacker to extract information [198]. Further work on securing GPUs to allow containers to run securely while making use of them would greatly increase the potential applications of BlackBox and secured cloud computing.

Although BlackBox is designed to work using existing hardware virtualization support, the upcoming Armv9 architecture, with its inclusion of the Arm Confidential Compute Architecture (CCA) [172], offers alternative mechanisms that can be used for implementing BlackBox. CCA's concept of Realms offers an alternative solution to providing PPASes and supporting the CSM. With CCA, Realms are supported by a separate Realm World alongside Arm's existing secure and non-secure worlds, complete with Arm's existing three privilege levels, EL0-3. As with the existing secure world, Realm World has access to both its own memory and the memory within the non-secure world. Realms are managed using the Realm Management Monitor (RMM) running in EL2 within the Realm World, giving it full access to Realm memory and CPU state as well as control over their execution. Realm execution and memory delegation is provided to the other worlds through the Realm Management Interface (RMI). CCA offers dynamically adjustable protected memory enabling PPASes to be implemented as Realms instead of separate NPTs. It may be possible for the current functionality of the CSM to be incorporated into the RMM. The

OS's exception vector table could be modified to use new RMI commands enabling the combined RMM/CSM to interpose on all interactions between the OS and containers similar to BlackBox's existing design. The CSM hypercall ABI, could be added as additional RMI commands. Given that many of the CSM's functionalities are OS-specific, an interesting research challenge will be exploring whether a combined RMM/CSM can be designed in a way that is capable of efficiently supporting multiple OSes with only minor OS modifications.

While the codebase of BlackBox is small enough that it could be formally verified actually doing so remains an open problem. SeKVM [170] was successfully formally verified and is similar in design and of comparable size to BlackBox, which provides some evidence that formal verification of BlackBox is possible. A formally verified implementation of BlackBox would offer greater guarantees of security to apps running in protected containers and potentially enable BlackBox to run in more application areas that require a high degree of operational guarantees. However, while similar in design to SeKVM, there are several significant differences that will require further innovations to enable formal verification of BlackBox. Most notably, while SeKVM is a hypervisor with a narrow interface, the CSM must support the broad system call interface of a commodity OS. A key challenge is how to verify not only the security of the CSM, but the functional correctness of the CSM in terms of ensuring that it preserves the existing semantics of system calls.

References

[1] S. Chiueh and N. Tzi-cker, "A Survey on Virtualization Technologies," *Rpe Report*, vol. 142, Jun. 2005.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, Bolton Landing, NY, Oct. 2003, pp. 164–177.

[3] C. Dall and J. Nieh, "KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*, Salt Lake City, UT, Mar. 2014, pp. 333–347.

[4] E. Bugnion, J. Nieh, and D. Tsafrir, *Hardware and Software Support for Virtualization* (Synthesis Lectures on Computer Architecture). Morgan and Claypool Publishers, Feb. 2017.

[5] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environments," in *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, Dec. 2002, pp. 361–376.

[6] T. L. Foundation, *Linux Containers - LXC - Introduction*, `https://linuxcontainers.org/lxc/introduction/`, 2022.

[7] *OpenVZ Linux Containers*, `http://www.openvz.org`, 2014.

[8] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom, "The Use of Name Spaces in Plan 9," in *Proceedings of the 5th Workshop on ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring*, ser. EW 5, Mont Saint-Michel, France, Sep. 1992, pp. 1–5.

[9] Docker, Inc., *Empowering App Development for Developers - Docker*, `https://www.docker.com`, 2021.

[10] C. N. C. Foundation, *Kubernetes*, `https://kubernetes.io`, 2023.

[11] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh, "Cells: A Virtual Mobile Smartphone Architecture," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, Cascais, Portugal, Oct. 2011, pp. 173–187.

[12] A. V. Hof, H. Jamjoom, J. Nieh, and D. Williams, "Flux: Multi-Surface Computing in Android," in *Proceedings of the 7th European Conference on Computer Systems (EuroSys 2015)*, Bordeaux, France, Apr. 2015, 24:1–17.

[13] A. V. Hof and J. Nieh, "AnDrone: Virtual Drone Computing in the Cloud," in *Proceedings of the 11th European Conference on Computer Systems (EuroSys 2019)*, Dresden, Germany, Mar. 2019, 6:1–16.

[14] A. V. Hof and J. Nieh, "BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, Carlsbad, CA, Jul. 2022, pp. 683–700.

[15] K. Truong, *INFOGRAPHIC: Users weighed down by multiple gadgets – survey reveals the most carried devices*, `https://nakedsecurity.sophos.com/2013/03/14/devices-wozniak-infographic/`, Mar. 2013.

[16] S. J. Yang, J. Nieh, M. Selsky, and N. Tiwari, "The Performance of Remote Display Mechanisms for Thin-Client Computing," in *Proceedings of the 2002 USENIX Annual Technical Conference (USENIX ATC 2002)*, Monterey, CA, Jun. 2002, pp. 131–146.

[17] A. Lai and J. Nieh, "Limits of Wide-Area Thin-Client Computing," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2002)*, Marina del Rey, CA, Jun. 2002, pp. 228–239.

[18] J. Nieh, S. J. Yang, and N. Novik, "Measuring Thin-Client Performance Using Slow-Motion Benchmarking," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 1, pp. 87–115, Feb. 2003.

[19] S. J. Yang, J. Nieh, S. Krishnappa, A. Mohla, and M. Sajjadpour, "Web Browsing Performance of Wireless Thin-Client Computing," in *Proceedings of the 12th International World Wide Web Conference (WWW 2003)*, Budapest, Hungary, May 2003, pp. 68–79.

[20] A. Lai, J. Nieh, B. Bohra, V. Nandikonda, A. P. Surana, and S. Varshneya, "Improving Web Browsing on Wireless PDAs Using Thin-Client Computing," in *Proceedings of the 13th International World Wide Web Conference (WWW 2004)*, New York, NY, May 2004, pp. 143–154.

[21] R. Baratto, S. Potter, G. Su, and J. Nieh, "MobiDesk: Mobile Virtual Desktop Computing," in *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking (MobiCom 2004)*, Philadelphia, PA, Sep. 2004, pp. 1–15.

[22] R. Baratto, L. Kim, and J. Nieh, "THINC: A Virtual Display Architecture for Thin-Client Computing," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, Brighton, UK, Oct. 2005, pp. 277–290.

[23] J. Kim, R. Baratto, and J. Nieh, "pTHINC: A Thin-Client Architecture for Mobile Wireless Web," in *Proceedings of the 15th International World Wide Web Conference (WWW 2006)*, Edinburgh, Scotland, May 2006, pp. 143–152.

[24] Apple Inc., *Apple Airplay - Play Content from iOS Devices on Apple TV.* `https://www.apple.com/airplay/`, 2015.

[25] Google Inc., *Chromecast*, `https://www.google.com/chrome/devices/chromecast/`, 2014.

[26] A. Venkatraman, *CIOs Distrust Public Cloud for Mission-Critical Work, says IDC*, `http://www.computerweekly.com/news/2240170818/CIOs-distrust-public-cloud-for-mission-critical-work-says-IDC`, Nov. 2012.

[27] C. Page, *Cloud Computing Voted as Fundamentally Insecure*, `http://www.theinquirer.net/inquirer/news/2381817/cloud-computing-voted-as-fundamentally-insecure`, Nov. 2014.

[28] OpenSignal, *Android Fragmentation Visualized*, `http://opensignal.com/reports/2014/android-fragmentation/`, Aug. 2014.

[29] J. Onorato, *Modifying AIDL Compiler to Generate C++ Code*, `https://groups.google.com/forum/#\protect\protect\leavevmode@ifvmode\kern-.1667em\relaxtopic/android-framework/i0SWc9cHEJ0`, Nov. 2011.

[30] OpenVZ, *Checkpoint/Restore in Userspace*, `http://www.criu.org`, 2018.

[31] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters," *Journal of Physics: Conference Series*, vol. 46, pp. 494–499, Sep. 2006.

[32] W. R. Dieter and J. E. Lumpp Jr, "User-Level Checkpointing for LinuxThreads Programs," in *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX ATC 2001)*, Berkeley, CA, Jun. 2001, pp. 81–92.

[33] O. Laadan and J. Nieh, "Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems," in *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX ATC 2007)*, Santa Clara, CA, Jun. 2007, pp. 323–336.

[34] S. Kazemi, R. Garg, and G. Cooperman, "Transparent Checkpoint-Restart for Hardware-Accelerated 3D Graphics," *CoRR*, vol. abs/1312.6650, Dec. 2013.

[35] M. R. Hines and K. Gopalan, "Post-Copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging and Dynamic Self-Ballooning," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2009)*, Washington, DC, Mar. 2009, pp. 51–60.

[36] Aurora Software, *Quadrant Standard*, `http://www.aurorasoftworks.com/products/quadrant`, 2014.

[37] Apple Inc., *SunSpider 1.0.2*, `https://www.webkit.org/perf/sunspider/sunspider.html`, 2014.

[38] N. Viennot, E. Garcia, and J. Nieh, "A Measurement Study of Google Play," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2014)*, Austin, TX, Jun. 2014, pp. 221–233.

[39] R. Rashid and G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," in *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP 1981)*, Pacific Grove, CA, Dec. 1981, pp. 64–75.

[40] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba: a Distributed Operating System for the 1990s," *IEEE Computer*, vol. 23, no. 5, pp. 44–53, May 1990.

[41] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser, "Chorus Distributed Operating System," *Computing Systems*, vol. 1, no. 4, pp. 305–370, Apr. 1988.

[42] M. L. Powell and B. P. Miller, "Process Migration in DEMOS/MP," in *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP 1983)*, Bretton Woods, NH, Oct. 1983, pp. 110–119.

[43] A. Barak and O. Laádan, "The MOSIX Multicomputer Operating System for High Performance Cluster Computing," *Journal of Future Generation Computer Systems*, vol. 13, no. 4–5, pp. 361–372, Mar. 1998.

[44] F. Douglis and J. K. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software - Practice and Experience*, vol. 21, no. 8, pp. 757–785, Aug. 1991.

[45] D. R. Cheriton, "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, vol. 1, no. 2, pp. 19–42, Apr. 1984.

[46] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast Transparent Migration for Virtual Machines," in *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC 2005)*, Anaheim, CA, Apr. 2005, pp. 1–25.

[47] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI 2005)*, Boston, MA, May 2005, pp. 273–286.

[48] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis, "The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket?" *ACM SIGOPS Operating Systems Review*, vol. 44, no. 4, pp. 124–135, Dec. 2010.

[49] C. Dall, J. Andrus, A. V. Hof, O. Laadan, and J. Nieh, "The Design, Implementation, and Evaluation of Cells: A Virtual Mobile Smartphone Architecture," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 3, 9:1–31, Aug. 2012.

[50] J. S. Plank, "An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance," Dept. of Computer Science, University of Tennessee, Tech. Rep. UT-CS-97-372, Jul. 1997.

[51] E. Roman, "A Survey of Checkpoint/Restart Implementations," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-54942, Jul. 2002.

[52] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang, "Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Washington, DC, Apr. 2005, pp. 1–8.

[53] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing Under Unix," in *Proceedings of the 1995 USENIX Winter Technical Conference*, New Orleans, LA, Jan. 1995, pp. 213–223.

[54] T. Tannenbaum and M. Litzkow, "The Condor Distributed Processing System," *Dr. Dobb's Journal*, vol. 20, no. 227, pp. 40–48, 2 Feb. 1995.

[55] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the Library OS from the Top Down," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, Newport Beach, CA, Mar. 2011, pp. 291–304.

[56] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt, "Composing OS Extensions Safely and Efficiently with Bascule," in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys 2013)*, Prague, Czech Republic, Apr. 2013, pp. 239–252.

[57] B. A. Kingsbury and J. T. Kline, "Job and Process Recovery in a UNIX-based Operating System," in *Proceedings of the 1989 USENIX Winter Technical Conference*, San Diego, CA, Jan. 1989, pp. 355–364.

[58] C. R. Landau, "The Checkpoint Mechanism in KeyKOS," in *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, Dourdan, France, Sep. 1992, pp. 86–91.

[59] B. K. Schmidt, "Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches," Ph.D. dissertation, Stanford University, Aug. 2000.

[60] O. Laadan, D. Phung, and J. Nieh, "Transparent Checkpoint-Restart of Distributed Applications on Commodity Clusters," in *Proceedings of the 2005 IEEE International Conference on Cluster Computing (Cluster 2005)*, Boston, MA, Sep. 2005, pp. 1–13.

[61] A. Beguelin, E. Seligman, and P. Stephan, "Application Level Fault Tolerance in Heterogeneous Networks of Workstations," *Journal of Parallel and Distributed Computing*, vol. 43, no. 2, pp. 147–155, Jun. 1997.

[62] Y. Huang, C. Kintala, and Y. M. Wang, "Software Tools and Libraries for Fault Tolerance," *IEEE Bulletin of the Tecnnical Committee on Operating System and Application Environments*, vol. 7, no. 4, pp. 5–9, 1995.

[63] L. V. Kale and S. Krishnan, "CHARM++: a Portable Concurrent Object Oriented System Based on C++," in *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1993)*, Washington, DC, Sep. 1993, pp. 91–108.

[64] 0xlab, *Android DMTCP*, `http://github.com/0xlab`.

[65] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau, "Framework for Instruction-level Tracing and Analysis of Program Executions," in *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE 2006)*, Ottawa, Ontario, Canada, Jun. 2006, pp. 154–163.

[66] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling Dynamic Program Analysis from Execution in Virtual Environments," in *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX ATC 2008)*, Boston, MA, Jun. 2008, pp. 1–14.

[67] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers*, vol. 36, no. 4, pp. 471–482, Apr. 1987.

[68] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang, "R2: An Application-Level Kernel for Record and Replay," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, Berkeley, CA, Dec. 2008, pp. 193–208.

[69] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically Classifying Benign and Harmful Data Races Using Replay Analysis," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007)*, San Diego, CA, Jun. 2007, pp. 22–31.

[70] Y. Saito, "Jockey: a User-Space Library for Record-Replay Debugging," in *Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging (AADEBUG 2005)*, Monterey, CA, Sep. 2005, pp. 69–76.

[71] D. Subhraveti and J. Nieh, "Record and Transplay: Partial Checkpointing for Replay Debugging Across Heterogeneous Systems," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2011)*, San Jose, CA, Jun. 2011, pp. 109–120.

[72] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 1995)*, Copper Mountain, CO, Dec. 1995, pp. 1–11.

[73] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen, "Multi-stage Replay with Crosscut," in *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2010)*, Pittsburgh, PA, Mar. 2010, pp. 13–24.

[74] O. Laadan, N. Viennot, and J. Nieh, "Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2010)*, New York, NY, Jun. 2010, pp. 155–166.

[75] O. Laadan, N. Viennot, C.-C. Tsai, C. Blinn, J. Yang, and J. Nieh, "Pervasive Detection of Process Races in Deployed Systems," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, Cascais, Portugal, Oct. 2011, pp. 353–367.

[76] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting Past and Present Intrusions through Vulnerability-Specific Predicates," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, Brighton, United Kingdom, Oct. 2005, pp. 91–104.

[77] N. Viennot, S. Nair, and J. Nieh, "Transparent Mutable Replay for Multicore Debugging and Patch Validation," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, Houston, TX, Mar. 2013, pp. 127–138.

[78] U.S. Department of Transportation, *FAA Drone Registry Tops One Million*, `https://www.transportation.gov/briefing-room/faa-drone-registry-tops-one-million`, Jan. 2018.

[79] Amazon.com Inc., *Amazon Prime Air*, `https://www.amazon.com/Amazon-Prime-Air/b?node=8037720011`, 2016.

[80] United Parcel Service, *UPS Tests Residential Delivery Via Drone Launched From atop Package Car*, https://pressroom.ups.com/pressroom/ContentDetailsViewer.page?ConceptType=PressReleases&id=1487687844847-162, Feb. 2017.

[81] Deutsche Post AG, *Unmanned Aerial Vehicles in Logistics*, http://www.dhl.com/content/dam/downloads/g0/about_us/logistics_insights/dhl_trend_report_uav.pdf, 2014.

[82] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang, "Bringing Virtualization to the x86 Architecture with the Original VMware Workstation," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 4, 12:1–12:51, Nov. 2012.

[83] O. Laadan and J. Nieh, "Operating System Virtualization: Practice and Experience," in *Proceedings of the 3rd Annual Haifa Experimental Systems Conference (SYSTOR 2010)*, Haifa, Israel, May 2010, pp. 1–12.

[84] Google Inc., *Android Things*, https://developer.android.com/things/index.html, 2018.

[85] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh, "POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing," in *Proceedings of the 8th European Conference on Computer Systems (EuroSys 2016)*, London, UK, Apr. 2016, 19:1–17.

[86] Raspberry Pi Foundation, *Raspberry Pi 3 Model B*, https://www.raspberrypi.org/products/raspberry-pi-3-model-b/, 2017.

[87] Emlid Ltd., *Emlid Navio2 - Raspberry Pi Autopilot HAT Powered by ArduPilot & ROS*, https://emlid.com/navio/, 2017.

[88] E. Kim, J. Lee, and K. G. Shin, "Real-time Prediction of Battery Power Requirements for Electric Vehicles," in *Proceedings of the 4th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS 2013)*, Philadelphia, PA, Apr. 2013, pp. 11–20.

[89] A. Cappiello, I. Chabini, E. K. Nam, A. Lue, and M. A. Zeid, "A Statistical Model of Vehicle Emissions and Fuel Consumption," in *Proceedings of the 5th IEEE International Conference on Intelligent Transportation Systems*, Singapore, Sep. 2002, pp. 801–809.

[90] C. Tseng, C. Chau, K. M. Elbassioni, and M. Khonji, "Flight Tour Planning with Recharging Optimization for Battery-operated Autonomous Drones," *ArXiv*, vol. abs/1703.10049, Sep. 2017.

[91] R. D'Andrea, "Guest Editorial: Can Drones Deliver?" *IEEE Transactions on Automation Science and Engineering*, vol. 11, no. 3, pp. 647–648, Jul. 2014.

[92] K. Dorling, J. Heinrichs, G. G. Messier, and S. Magierowski, "Vehicle routing problems for drone delivery," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 47, no. 1, pp. 70–85, Jan. 2017.

[93] Qualcomm, *LTE Unmanned Aircraft Systems Trial Report*, `https://www.qualcomm.com/documents/lte-unmanned-aircraft-systems-trial-report`, May 2017.

[94] Qualcomm, *Leading the World to 5G: Evolving Cellular Technologies for Safer Drone Operation*, `https://www.qualcomm.com/media/documents/files/leading-the-world-to-5g-evolving-cellular-technologies-for-safer-drone-operation.pdf`, Sep. 2016.

[95] Linux Kernel Organization Inc, *Real-Time Linux Wiki*, `https://rt.wiki.kernel.org/index.php/Main_Page`, Aug. 2016.

[96] C. Dall, "The Design, Implementation, and Evaluation of the Linux ARM Hypervisor," Ph.D. dissertation, Columbia University, Feb. 2018.

[97] J. T. Lim, C. Dall, S.-W. Li, J. Nieh, and M. Zyngier, "NEVE: Nested Virtualization Extensions for ARM," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*, Shanghai, China, Oct. 2017, pp. 201–217.

[98] C. Dall, S.-W. Li, and J. Nieh, "Optimizing the Design and Implementation of the Linux ARM Hypervisor," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 2017)*, Santa Clara, CA, Jul. 2017, pp. 221–234.

[99] C. Dall, S.-W. Li, J. T. Lim, J. Nieh, and G. Koloventzos, "ARM Virtualization: Performance and Architectural Implications," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA 2016)*, Seoul, South Korea, Jun. 2016, pp. 304–316.

[100] S. Potter and J. Nieh, "Improving Virtual Appliance Management through Virtual Layered File Systems," in *Proceedings of the 25th Large Installation System Administration Conference (LISA 2011)*, Boston, MA, Dec. 2011, pp. 25–38.

[101] S. Potter and J. Nieh, "Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems," in *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC 2010)*, Boston, MA, Jun. 2010, pp. 103–116.

[102] S. Potter, "Virtualization Mechanisms for Mobility, Security and System Administration," Ph.D. dissertation, Columbia University, May 2010.

[103] J. Andrus, "Multi-Persona Mobile Computing," Ph.D. dissertation, Columbia University, Feb. 2015.

[104] ArduPilot, *MAVProxy*, https://ardupilot.github.io/MAVProxy/html/index.html, 2017.

[105] I. Vereninov, *Issue with Navio2 with a Potential Flyaway Condition*, https://community.emlid.com/t/issue-with-navio2-with-a-potential-flyaway-condition/2179, Mar. 2016.

[106] Da-Jiang Innovations, *Flame Wheel ARF Kit*, https://www.dji.com/flame-wheel-arf, 2018.

[107] T-Motor, *6th Anniversary Limited Edition T-Motor Combo Pack*, http://store-en.tmotor.com/goods.php?id=453, 2017.

[108] Raspberry Pi Foundation, *Camera Module V2 - Raspberry Pi*, https://www.raspberrypi.org/products/camera-module-v2/, 2016.

[109] HobbyKing, *Turnigy Nano-Tech 5000mah 3S 3570C Lipo Pack w/XT-90*, https://hobbyking.com/en_us/turnigy-battery-nano-tech-5000mah-3s-35-70c-lipo-pack-xt-90.html, 2017.

[110] Raspberry Pi Foundation, *Raspbian*, https://www.raspberrypi.org/downloads/raspbian/, 2018.

[111] Alpine Linux Development Team, *Alpine Linux*, https://alpinelinux.org/, 2019.

[112] ArduPilot, *ArduPilot Open Source Autopilot*, http://ardupilot.org/, 2019.

[113] PassMark Software Inc., *PassMark PerformanceTest - Android Apps on Google Play*, https://play.google.com/store/apps/details?id=com.passmark.pt_mobile, Oct. 2017.

[114] The Linux Foundation, *Cyclictest*, https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest, Jun. 2017.

[115] V. Gueant, *iPerf - The TCP, UDP and SCTP Network Bandwidth Measurement Tool*, https://iperf.fr/, 2015.

[116] A. Waterland, *Stress Project Page*, https://people.seas.harvard.edu/~apw/stress/, Jul. 2014.

[117] M. Arroyo, H. Kobayashi, S. Sethumadhavan, and J. Yang, "FIRED: Frequent Inertial Resets with Diversification for Emerging Commodity Cyber-Physical Systems," *ArXiv*, vol. abs/1702.06595, Feb. 2017.

[118] 3D Robotics, *DroneKit Log Analyzer Documentation*, http://la.dronekit.io/, 2015.

[119] Monsoon Solutions Inc., *Monsoon Solutions Power Monitor*, https://www.msoon.com/LabEquipment/PowerMonitor/, Aug. 2016.

[120] ArduPilot, *APM Planner 2 Home*, http://ardupilot.org/planner2/, 2019.

[121] J. Kos, *TX/RX Latency Test Results*, http://rc.runryder.com/helicopter/t172571p1/, 2015.

[122] J. Drew, *JR Module in Taranis - Latency Testing*, https://www.rcgroups.com/forums/showthread.php?2169724-JR-Module-in-Taranis-latency-testing, 2014.

[123] ArduPilot, *SITL Simulator (Software in the Loop)*, http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html, 2019.

[124] 3D Robotics, *DroneKit by 3D Robotics*, http://dronekit.io/, 2015.

[125] M. S. Gordon, J. R. Kozloski, P. K. Malkin, and C. A. Pickover, *Autonomous Drone Service System*, US Patent Application no. US20160306355A1, Apr. 2015.

[126] J. Yapp, R. Seker, and R. Babiceanu, "UAV as a Service: Enabling On-Demand Access and On-the-Fly Re-Tasking of Multi-tenant UAVs Using Cloud Services," in *Proceedings of the 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC 2016)*, Sacramento, CA, Sep. 2016, pp. 1–8.

[127] A. Koubaa, B. Qureshi, M.-F. Sriti, Y. Javed, and E. Tovar, "A Service-Oriented Cloud-Based Management System for the Internet-of-Drones," in *Proceedings of the 2017 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC 2017)*, Coimbra, Portugal, Apr. 2017, pp. 329–335.

[128] S. Y. M. Mahmoud and N. Mohamed, "Toward a Cloud Platform for UAV Resources and Services," in *Proceedings of the 4th IEEE Symposium on Network Cloud Computing and Applications (NCCA 2015)*, Munich, Germany, Jun. 2015, pp. 23–30.

[129] S. Mahmoud, N. Mohamed, and J. Al-Jaroodi, "Integrating UAVs into the Cloud Using the Concept of the Web of Things," *Journal of Robotics*, vol. 2015, 10:1–10:10, Jan. 2015.

[130] M. Chiaberge, A. Toma, G. Ermacora, S. Rosa, B. Bona, M. Silvagni, M. Gaspardone, and R. Antonini, "A Cloud Based Service for Management and Planning of Autonomous UAV Missions in Smart City Scenarios," in Sep. 2014, vol. 8906, pp. 20–26.

[131] G. Ermacora, S. Rosa, and A. Toma, "Fly4SmartCity: A Cloud Robotics Service for Smart City Applications," *Journal of Ambient Intelligence and Smart Environments*, vol. 8, no. 3, pp. 347–358, Apr. 2016.

[132] D. Vasisht, Z. Kapetanovic, J. Won, X. Jin, R. Chandra, S. Sinha, A. Kapoor, M. Sudarshan, and S. Stratman, "FarmBeats: An IoT Platform for Data-Driven Agriculture," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2017)*, Boston, MA, Mar. 2017, pp. 515–529.

[133] Galois Inc., *Update: FreeRTOS for Xen on ARM Systems*, `https://galois.com/blog/2016/07/update-freertos-xen-arm-systems/`, Jul. 2016.

[134] J. Daugherty, *Getting Started with FreeRTOS for Xen on ARM*, `https://blog.xenproject.org/2015/02/02/getting-started-with-freertos-for-xen-on-arm-2/`, Feb. 2015.

[135] Galois Inc., *SMACCMPilot*, `https://smaccmpilot.org/`, 2017.

[136] P. Bonzini, *Realtime KVM*, `https://lwn.net/Articles/656807/`, Sep. 2015.

[137] Intel Corporation, *Intel Software Guard Extensions Programming Reference*, `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`, Oct. 2014.

[138] ARM Ltd., *ARM Security Technology - Building a Secure System using TrustZone Technology*, `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`, Apr. 2009.

[139] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI 2014)*, Broomfield, CO, Oct. 2014, pp. 267–283.

[140] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Linda, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI 2016)*, Savannah, GA, Nov. 2016, pp. 689–703.

[141] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)*, Seattle, WA, Mar. 2008, pp. 2–13.

[142] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "InkTag: Secure Applications on an Untrusted Operating System," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, Houston, TX, Mar. 2013, pp. 265–278.

[143] J. Yang and K. G. Shin, "Using Hypervisor to Provide Data Secrecy for User Applications on a Per-Page Basis," in *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2008)*, Seattle, WA, Mar. 2008, pp. 71–80.

[144] Y. Cheng, X. Ding, and R. H. Deng, "Efficient Virtualization-Based Application Protection Against Untrusted Operating System," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIA CCS 2015)*, Singapore, Mar. 2015, pp. 345–356.

[145] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel, "Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2016)*, Atlanta, GA, May 2016, pp. 277–290.

[146] U. Steinberg and B. Kauer, "NOVA: A Microhypervisor-based Secure Virtualization Architecture," in *Proceedings of the 5th European Conference on Computer Systems (EuroSys 2010)*, Paris, France, Apr. 2010, pp. 209–222.

[147] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, P. chung Yew, and W. Mao, "Tamper-Resistant Execution in an Untrusted Operating System Using A Virtual Machine Monitor," Parallel Processing Institute, Fudan University, Technical Report PPITR-2007-08001, Aug. 2007.

[148] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end Arguments in System Design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 277–288, Nov. 1984.

[149] S. Landau, "Making Sense from Snowden: What's Significant in the NSA Surveillance Revelations," *IEEE Security and Privacy*, vol. 11, no. 4, pp. 54–63, Jul. 2013.

[150] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, Chicago, IL, Nov. 2009, pp. 199–212.

[151] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM Side Channels and Their Use to Extract Private Keys," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012)*, Raleigh, NC, Oct. 2012, pp. 305–316.

[152] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (IEEE S&P 2015)*, San Jose, CA, May 2015, pp. 591–604.

[153] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-Tenant Side-Channel Attacks in Paas Clouds," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS 2014)*, Scottsdale, AZ, Nov. 2014, pp. 990–1003.

[154] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks Are Practical," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (IEEE S&P 2015)*, San Jose, CA, May 2015, pp. 605–622.

[155] M. Backes, G. Doychev, and B. Kopf, "Preventing Side-Channel Leaks in Web Traffic: A Formal Approach," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS 2013)*, San Diego, CA, Feb. 2013, pp. 1–17.

[156] International Organization for Standardization and International Electrotechnical Commission, *ISO/IEC 11889-1:2015 - Information technology – Trusted Platform Module Library*, `https://www.iso.org/standard/66510.html`, Sep. 2016.

[157] Google, *HTTPS Encryption on the Web – Google Transparency Report*, `https://transparencyreport.google.com/https/overview`, Apr. 2018.

[158] Let's Encrypt, *Let's Encrypt Stats - Let's Encrypt*, `https://letsencrypt.org/stats/`, Apr. 2018.

[159] S. Hykes, *Introducing runC: A lightweight Universal Container Runtime*, `https://www.docker.com/blog/runc/`, Jun. 2015.

[160] P. Stewin and I. Bystrov, "Understanding DMA Malware," in *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2012)*, Heraklion, Crete, Greece, Jul. 2013, pp. 21–41.

[161] Amazon Web Services, Inc., *Building an Enclave Image File*, `https://docs.aws.amazon.com/enclaves/latest/user/building-eif.html`, 2021.

[162] Samsung Electronics Company, Ltd., *Samsung Knox - White Paper*, `https://docs.samsungknox.com/admin/whitepaper/kpe/samsung-knox.htm`, 2021.

[163] S. Checkoway and H. Shacham, "Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, Houston, TX, Mar. 2013, pp. 253–264.

[164] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security*, vol. 15, no. 1, pp. 1–34, Mar. 2012.

[165] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, Alexandria, VA, Oct. 2007, pp. 552–561.

[166] R. Cui, L. Zhao, and D. Lie, "Emilia: Catching Iago in Legacy Code," in *Proceedings of the 2021 ISOC Network and Distributed Systems Security Symposium (NDSS 2021)*, Virtual Event, Feb. 2021, pp. 1–17.

[167] C. Dall, S.-W. Li, J. T. Lim, and J. Nieh, "ARM Virtualization: Performance and Architectural Implications," *ACM SIGOPS Operating Systems Review*, vol. 52, no. 1, pp. 45–56, Jul. 2018.

[168] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A Verified Modern Cryptographic Library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, Dallas, TX, Oct. 2017, pp. 1789–1806.

[169] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "A Secure and Formally Verified Linux KVM Hypervisor," in *Proceedings of the 2021 IEEE Symposium on Security and Privacy (IEEE S&P 2021)*, San Francisco, CA, May 2021, pp. 1782–1799.

[170] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021)*, Vancouver, BC Canada, Aug. 2021, pp. 3953–3970.

[171] R. Tao, J. Yao, X. Li, S.-W. Li, J. Nieh, and R. Gu, "Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware," in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP 2021)*, Virtual Event, Germany, Oct. 2021, pp. 866–881.

[172] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell, "Design and Verification of the Arm Confidential Compute Architecture," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, Carlsbad, CA, Jul. 2022, pp. 465–484.

[173] L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," in *Proceedings of the 1996 USENIX Annual Technical Conference (USENIX ATC 1996)*, San Diego, CA, Jan. 1996, pp. 279–294.

[174] R. Russell, *Hackbench*, `http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c`, Jan. 2008.

[175] The Apache Software Foundation, *ab - Apache HTTP Server Benchmarking Tool*, `http://httpd.apache.org/docs/2.4/programs/ab.html`, Apr. 2015.

[176] Redis Labs, *memtier_benchmark*, `https://github.com/RedisLabs/memtier_benchmark`, Apr. 2015.

[177] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC 2010)*, Indianapolis, IN, Jun. 2010, pp. 143–154.

[178] R. Jones, *Netperf*, `https://github.com/HewlettPackard/netperf`, Jun. 2018.

[179] T. Hromatka, *RFE: Use a cBPF Binary Tree for Large Seccomp Filters*, `https://github.com/seccomp/libseccomp/issues/116`, Apr. 2018.

[180] LTP developers, *LTP - Linux Test Project*, `https://linux-test-project.github.io/`, 2021.

[181] C. che Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 2017)*, Santa Clara, CA, Jul. 2017, pp. 645–658.

[182] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020)*, Lausanne, Switzerland, Mar. 2020, pp. 955–970.

[183] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. R. Pietzuch, "SGX-LKL: Securing the Host OS Interface for Trusted Execution," *ArXiv*, vol. abs/1908.11143, Aug. 2019.

[184] V. A. Sartakov, L. Vilanova, and P. Pietzuch, "CubicleOS: A Library OS with Software Componentisation for Practical Isolation," in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*, Virtual Event, Apr. 2021, pp. 546–558.

[185] Z. Hua, Y. Yu, J. Gu, Y. Xia, H. Chen, and B. Zang, "TZ-Container: Protecting Container From Untrusted OS with ARM TrustZone," *Science China Information Sciences*, vol. 64, pp. 1869–1919, Aug. 2021.

[186] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys 2017)*, Niagara Falls, NY, Jun. 2017, pp. 488–501.

[187] A. M. Azab, P. Ning, and X. Zhang, "SICE: A Hardware-Level Strongly Isolated Computing Environment for X86 Multi-Core Platforms," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, Chicago, IL, Oct. 2011, pp. 375–388.

[188] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB Reduction and Attestation," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy (IEEE S&P 2010)*, Oakland, CA, May 2010, pp. 143–158.

[189] G. Heiser and B. Leslie, "The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors," in *Proceedings of the 1st ACM Asia-pacific Workshop on Workshop on Systems (APSys 2010)*, New Delhi, India, Aug. 2010, pp. 19–24.

[190] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "SeL4: Formal Verification of an OS Kernel," in *Proceedings of the ACM 22nd Symposium on Operating Systems Principles (SOSP 2019)*, Big Sky, MO, Oct. 2009, pp. 207–220.

[191] S.-W. Li, J. S. Koh, and J. Nieh, "Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*, Santa Clara, CA, Aug. 2019, pp. 1357–1374.

[192] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon, "X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019)*, Providence, RI: Association for Computing Machinery, Apr. 2019, pp. 121–135.

[193] Google, Inc., *gVisor: Application Kernel for Containers*, `https://github.com/google/gvisor`, 2021.

[194] Google, Inc., *Language Focused Docker Images, Minus the Operating System*, `https://github.com/GoogleContainerTools/distroless`, 2021.

[195] Linux Container Hardening Project, *Linux Container Hardening: A Project Focused on Future Proofing the Linux Kernel Primitives That Are Used to Make Containers*, `https://containerhardening.org/`, 2021.

[196] L. Vu, H. Sivaraman, and R. Bidarkar, "GPU Virtualization for High Performance General Purpose Computing on the ESX Hypervisor," in *Proceedings of the High Performance Computing Symposium (HPC 2014)*, Tampa, FL, Apr. 2014.

[197] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous Isolated Execution for Commodity GPUs," in *Proceedings of the 24th International Conference on Ar-*

chitectural Support for Programming Languages and Operating Systems (ASPLOS 2019), Providence, RI, Apr. 2019, pp. 455–468.

[198] T. Hunt, Z. Jia, V. Miller, A. Szekely, Y. Hu, C. J. Rossbach, and E. Witchel, "Telekine: Secure computing with cloud GPUs," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2020)*, Santa Clara, CA, Feb. 2020, pp. 817–833.