This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2024.3349561



# Supporting Flexible and Transparent User Interface Distribution across Mobile Devices

Sangeun Oh, Ahyeon Kim, Sunjae Lee,
Kilho Lee, Dae R. Jeong, Steven Y. Ko, and Insik Shin, *Member, IEEE*

***Abstract*— The growing trend of multi-device ownerships creates opportunities to use applications across devices. However, the current methods of app development/usage remain in the single-device paradigm, which is far below user expectations. For example, it is currently impossible for users to dynamically partition an existing app across different devices to utilize multiple surfaces. We introduce FLUID, a novel multi-device platform that supports simultaneous operation of multiple devices. FLUID aims to *i)* distribute the user interfaces (UIs) of a single app across multiple devices, *ii)* support unmodified legacy apps without extra engineering, and *iii)* support numerous apps with customized UIs. Previous approaches, like screen mirroring and app migration, do not satisfy those goals altogether. However, FLUID is designed to satisfy the goals. It can efficiently deploy UI objects to different devices by identifying only UI states necessary for accurate rendering. And FLUID can execute the distributed UI objects by supporting cross-device method invocations transparently and synchronizing the replicated UIs across devices. Furthermore, FLUID automatically handles unexpected events that may degrade its usability by efficiently maintaining the distributed UIs up to date. Our evaluation using 20 legacy apps shows that FLUID can transparently support numerous apps and is fast enough for interactive use.**

***Index Terms*—Mobile and Ubiquitous Systems, Multi-device Mobile Platforms, Multi-surface Computing, User Interface Distribution**

## I. Introduction

THE recent rapid development of IoT technologies has changed our lives completely. Various smart devices such as smartphones, tablets, home appliances, and even cars have proliferated, and users now typically own multiple devices. According to a survey [1], each US household has an average of 11 connected devices, with seven being equipped with screens of various sizes and shapes. Such a trend can change user–device interaction; that is, in the near future, users will be able to interact with a single application using multiple screens (i.e., multiple surfaces) on different devices. Therefore, it is unsurprising to foresee the paradigm shift of mobile device usage from *single-surface computing* to *multi-surface computing* in the era of IoT.

This work was supported in part by the National Research Foundation of Korea (NRF) grants (NRF-2021R1F1A1063785, NRF-2021R1A4A1032252, and NRF-2018R1A5A1059921), and the Institute for Information & Communications Technology Planning & Evaluation (IITP) grant (IITP-2023-2018-0-01431 and IITP-2023-RS-2022-00156360). Steven Y. Ko was supported in part by the NSERC Discovery grant (RGPIN/04061-2021).

S. Oh is with the Department of Software and Computer Engineering, Ajou University, Suwon 16499, South Korea (e-mail: sangeunoh@ajou.ac.kr).

A. Kim, S. Lee, D. R. Jeong, and I. Shin are with the School of Computing, KAIST, Daejeon 34141, South Korea (e-mail: {nonnos, sunjae1294, dae.r.jeong, insik.shin}@kaist.ac.kr).

K. Lee is with the School of AI Convergence and Dept. of Intelligent Semiconductors, Soongsil University, Seoul 06978, South Korea (e-mail: khlee.cs@ssu.ac.kr).

S. Y. Ko is with the School of Computing Science, Simon Fraser University, Burnaby V5A 1S6, BC, Canada (e-mail: steveyko@sfu.ca).

Insik Shin is a corresponding author.

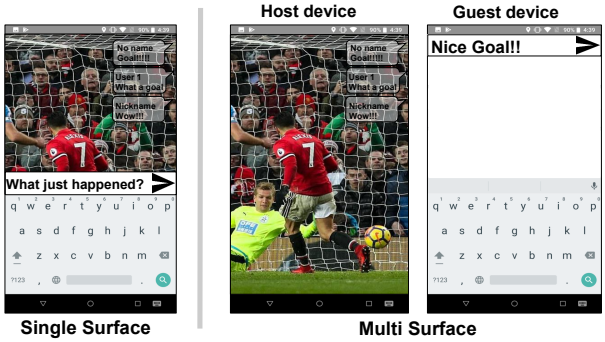



Fig. 1. Example scenario for a live streaming app. Multi-surface computing allows a user to enjoy a live broadcast on a full screen without overlapping with a keyboard UI.

With the ownership of multiple devices, many useful and interesting use cases can be envisioned for multi-surface computing. Specifically, such interaction environments can be more attractive owing to the following three driving factors. *i*) *Multi-function:* Various functionalities provided by a single app can be placed on multiple devices, thereby facilitating the simultaneous use of several functionalities. For example, as shown in Fig. 1, if a user can deploy the chatting function of a live streaming app to a different device, the user can enjoy both watching live broadcasting and communicating with other viewers more conveniently at the same time. *ii*) *Multi-device:* Various tasks can have different preferences for devices depending on their use because each device has distinct form factors. For instance, a user can control a video progress bar using their smartphone while watching a movie on the big screen of a smart TV. This provides easier and richer controls compared to using a TV remote controller. *iii*) *Multi-user:* Multiple users can work together on their own devices to easily accomplish a single task. For example, to reserve flight tickets for a group of colleagues, a user can distribute input fields to fill in personal information on the devices of their colleagues. The colleagues can then easily offer the necessary information (e.g., their passport number) on their own devices.

Unsurprisingly, there are existing solutions to support such multi-surface computing, which can be classified into four categories. The first is to use custom apps created for specific multi-surface usages, such as continuous video streaming between devices (e.g., Netflix) and joint document editing among users (e.g., Google Docs). However, as this approach was only supported by custom apps, its applicability is extremely limited. Second, previous studies [2], [3] have proposed new

programming models and/or development tools that add multi-surface operations to existing apps to reduce the engineering costs. However, they have limited applicability because they cannot maintain the same look-and-feel for app-specific custom UIs. It is common practice to employ customized UIs when developing mobile applications. In our analysis of the top one hundred applications downloaded from the Google Play Store, conducted in May 2018, we observed that all apps customized their own UIs in an app-specific manner. Third, screen sharing (or mirroring) or app migration can be utilized to support unmodified legacy apps. The former is a method of duplicating the screen of one device on another device, usually with a larger screen or a higher resolution (e.g., AirPlay [4], Chromecast [5]), whereas the latter is a method for moving an app process to another device upon runtime to employ a different display [6]. However, they come with limited flexibility such that an app cannot fully exploit the advantages of multi-surface environments. In particular, through a screen-sharing technique, apps can display only the same screen content on multiple surfaces. This is because screen sharing is not provided at a more fine-grained level, such as displaying different UI elements on external devices. On the other hand, the app migration can allow an app to employ only one surface at a time instead of using multiple surfaces together. This indicates that it is impossible for an app to separate a video stream and a video progress bar to a smart TV and smartphone, respectively, using app migration.

In this paper, we introduce FLUID (**FL**exible **UI** **D**istribution)[1], a novel multi-surface platform that performs beyond the limitations of previous solutions. Our system has the following design requirements. *i*) *Flexibility:* It should support a fine-grained unit of distribution across devices so that users can utilize multiple surfaces with maximum flexibility. *ii*) *Ease of development (transparency):* It should support unmodified legacy apps and not impose any extra overhead for creating multi-surface apps, as compared to single-device app development. *iii*) *Applicability:* It should support a broad range of applications, including apps with various customized UI elements. *iv*) *Responsiveness:* The responsiveness must be high to handle user inputs promptly on multiple devices. Note that users expect responses to their inputs to be offered within 50 to 200 ms on an average [7]–[9].

To meet the above design requirements, FLUID enables the migration or replication of individual UI elements, dynamically selected by users, from one device (the host) to different devices (guests). Users can then interact with the UI elements on all or some of the devices. Accordingly, we address the following five technical challenges. i) *UI partitioning and distribution*: FLUID identifies a minimal-yet-complete set of UI objects and their associated graphical states necessary to render selected UI elements by carefully analyzing the UI source code of both the Android platform and target app. Subsequently, based on the analysis results, FLUID distributes the selected UIs to multiple devices. Applying local rendering on each device is suitable for mobile wireless environments because it reduces the network bandwidth required for UI distribution and considerably minimizes the number of round-trips in the network. ii) *Transparent distributed UI execution*: Even if the selected UIs are distributed, the target app must be executed completely as before by maintaining cooperation between distributed states (UI objects and graphical states) and remaining states. To achieve this, FLUID changes the local method calls to remote procedure calls (RPCs) to support transparent cross-device execution for existing apps. iii) *UI state synchronization:* When a user replicates a single UI element across different devices and interacts with it simultaneously, FLUID guarantees the overall accuracy of the replicated UI elements by synchronizing all relevant UI states. It enables FLUID to offer an option of replicating UIs for greater flexibility. iv) *Runtime change handling:* When the UI layout of an app changes owing to changes in the device configuration, distributed UIs should accordingly be able to maintain their states up to date. To this end, FLUID automatically identifies new UIs visually similar to the distributed UIs through their pixel similarity and redeploys them to the guest device with a small data transmission. v) *Seamless network fault handling:* When a network fault occurs during a multi-surface interaction, it should be properly handled to avoid any unexpected side effects on both the host and guest devices. FLUID seamlessly restores the distributed UIs on the host device without disrupting their execution flows.

To validate the concept of FLUID, we implemented an Android-based FLUID prototype using Google Pixel XL smartphones and Pixel C tablets. Through an evaluation using 20 legacy apps, we show that FLUID can support various new multi-surface use cases while providing high flexibility and full transparency. In addition, our network and power consumption experiments show that FLUID can significantly optimize network usage compared to different approaches (e.g., screen mirroring and app migration) for high responsiveness, thereby reducing power consumption.

## II. Use Cases

This section presents three categories of use cases to describe how FLUID can provide users with better user experiences. The current working prototype of FLUID supports these use cases using unmodified legacy apps.

**Better usability.** FLUID can provide a useful method to utilize multiple surfaces under various scenarios because it enables users to dynamically deploy UI elements to suitable devices in terms of usability, based on the screen sizes, available input methods, and other factors. For example, with most live-streaming apps, users can watch a live broadcast using a video UI while chatting with other viewers using a chat UI. However, at this time, because the chat UI with a virtual keyboard is usually placed over the video UI, inconvenient situations can often occur in which more than half of the video UI is covered on a smartphone display. This can significantly reduce the usability of the app, and users might not see important scenes, such as a scoring play in a live soccer broadcast. FLUID allows a user to enjoy chatting on one device while watching live broadcasts on another device by distributing the video and chat UIs to different devices. In addition, there are many difficult tasks to perform when using only a single screen, such as precisely controlling a video player's progress bar with a smart TV remote or entering a destination on a navigation app's search input box from the rear seat of a car. Under these situations, FLUID enables a

[1]See https://www.youtube.com/watch?v=uyoOW6Pmunw for our demo video.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2024.3349561



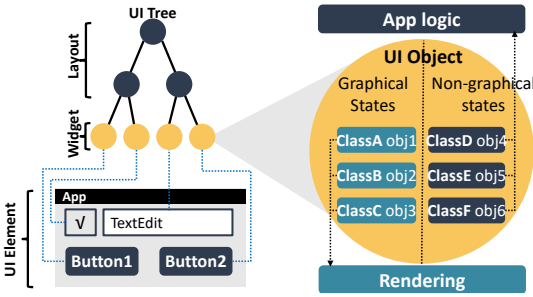



Fig. 2. Android UI framework

user to simply replicate the progress bar or the input box to a smartphone to utilize the benefits of the handheld device.

**Collaborative use.** In general, it is uncommon for multiple users to collaborate on a single task because of the limitation of output sharing (e.g., screen sharing [5], [10], [11]). Only a few cloud-based web applications (e.g., Google Docs) provide input collaboration features that allow multiple users to simultaneously provide input for the same task. However, FLUID facilitates transformation of an Android app into a collaborative app to support both output sharing and input collaboration. For instance, when entering personal information to reserve flight tickets for a group of colleagues, it could be extremely inconvenient for each colleague to enter their information in sequence on a single device (e.g., as the Expedia app expects [12]). FLUID allows a user to deploy input boxes for personal information to each colleague's device and enter the information parallelly.

**Privacy protection.** In many cases, there are risks of private information being leaked while sharing information across multiple devices through screen mirroring or app-level sharing. For instance, when mirroring a smartphone screen to a smart TV in a meeting room, a user may have to open a pattern lock of the smartphone or log into an application. Under such a situation, the user may unintentionally expose his lock pattern or password to other colleagues through a mirrored screen. As another example, when viewing a specific photograph or email content with colleagues on a smart TV, a user may have to expose all photographs or a full email list as well. With FLUID, the user can selectively distribute specific UI elements to show a specific photograph or email content to the smart TV while protecting the remaining UI elements (such as the list UI of emails or photographs) on the user's smartphone.

## III. Background

We target Android apps based on graphical user interfaces (GUIs) to realize the design space for FLUID. Therefore, the Android UI framework operation must be understood deeply, particularly the organization and rendering of UI elements of Android apps. In this section, we describe a brief background while defining some terminology used in our paper.

**UI architecture.** Fig. 2 illustrates that a GUI-based application contains various UI elements, such as buttons and text input boxes; one UI element is the smallest logical unit through which users can interact with an app from a user's perspective. Android provides two types of UI objects, *widgets* and *layouts*, and manages them in a single tree structure (called a *UI tree*) for each app. A widget is a graphical object connected to a UI element (e.g., a button) through one-to-one mapping, and the layout is a container object that determines how child UI objects are located on their screen. Each UI object has two categories of states: *i) graphical* states, which are the data used during the UI rendering process of Android (e.g., text, image, color, and animation), and *ii) non-graphical* states, which include the remaining data primarily used by app logic (e.g., event listeners), not related to rendering. Note that the graphical states are typically represented as memory objects, which are particularly defined as *UI resource objects* (*UI resources*). Thus, each UI element is logically associated with a collection of UI objects and their UI resources.

**UI thread.** It is a special thread that manages the UI tree while updating the states of all UI objects in each app. For instance, when a user presses a button, the UI thread executes an event handler connected to the UI object. The event handler then updates some graphical states (e.g., color) of the button or other UI objects. Note that these actions are typically performed using local method invocations, following the OOP encapsulation principle.

**Renderer thread.** This is a special thread that renders all UI objects in each app. When the UI thread completes updates of all UI objects and requests to render them, the renderer thread executes some methods related to rendering (e.g., `measure()`, `layout()`, and `draw()`) while traversing the UI tree from its root to leaf nodes. We describe this process in more detail in Section V-A.

## IV. FLUID: System Overview

To support the multi-surface scenarios described in Section II, we propose FLUID, which is a system-level solution that enables users to interact with a single unmodified app by utilizing multiple surfaces simultaneously. We adopt a UI element, which is the smallest logical unit through which users can interact with an app, as the unit of distribution to flexibly utilize multiple surfaces. This section presents the workflow and system design overview of FLUID.

### A. Workflow

Figure 3 illustrates the three-phase workflow of FLUID.

**Pairing phase.** First, FLUID forms a pairing among trusted devices (host and guest devices). The host device discovers and lists connectable guest devices nearby. The user selects the desired guest devices from the list along with a set of target apps for multi-surface interactions. The host device then securely establishes a network connection with the guests and transmits package files (APKs) of the target apps to the guests. Note that as long as the APK files are not updated on the host device, such a transfer process is not required.

**UI distribution phase.** FLUID provides an intuitive interface using multi-finger-tapping gestures[2], allowing a user to directly select UI elements to be distributed from a single app. When a user triggers the UI distribution for a host app, FLUID splits its UI tree into two parts: a subtree that includes the UI elements chosen by the user and the remaining UI tree. The subtree consists of a UI layout and widget objects along with their graphical resources. Upon completing the UI tree partition, FLUID forwards the selected subtree to each

[2]Our demonstration video shows how a user triggers a UI distribution.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2024.3349561



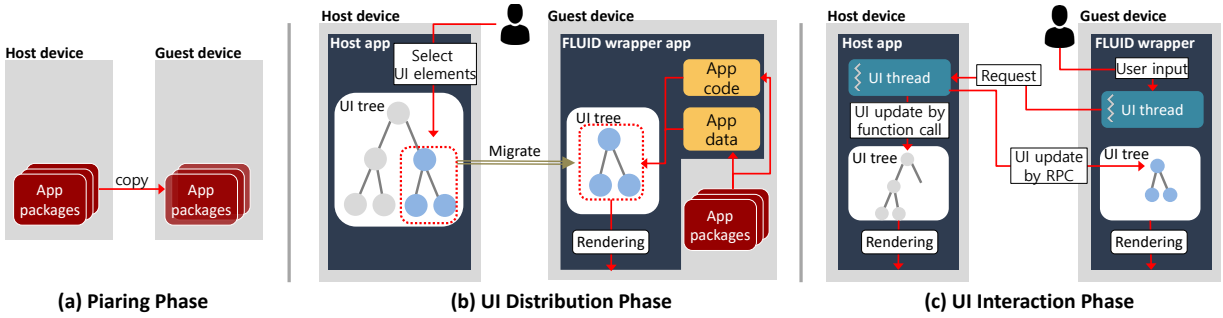



Fig. 3. FLUID architecture and workflow overview

guest device. On a guest device, FLUID launches a generic *wrapper* app to re-create the received UI subtree and related UI resources and displays the corresponding UIs with the same look-and-feel as the host side through local rendering (called as *guest* UI elements).

**UI interaction phase.** When guest UIs are successfully displayed on the guest surface, FLUID enables the user to interact with the host app by utilizing host and guest UIs concurrently, as if all UI elements are placed on the same device. For example, the user can control the video progress bar on a guest device to search for a specific scene of a video clip to be displayed on a host device. This scenario requires host-side app logic and guest-side UI objects to cooperate with each other across devices. Therefore, FLUID supports such cross-device cooperation by transforming local function calls to cross-device RPCs in a transparent manner.

### B. System Design

We designed FLUID to support the multi-surface execution model by solving the following challenges:

C1. How can UI objects be split and distributed efficiently while minimizing communication between devices?
C2. How can we support the interaction between app logic and the distributed UI objects in a transparent manner?
C3. How to guarantee app consistency when replicating UI objects across multiple surfaces?
C4. How to keep guest UIs up-to-date when a host app replaces the existing UI layout with a new layout because of changes to the device configuration?
C5. How to deal with unexpected network disconnections between devices during multi-surface interactions?

**C1.** As shown in Fig. 4, the primary design idea of FLUID is to split and distribute UI objects while minimizing communication cost between devices. We observed that a rendering process is frequently applied (e.g., 20–30 fps), and its result includes a considerable amount of pixel data. Based on this observation, we designed FLUID to display guest UI elements through local rendering on a guest device. This design enables rapid UI distribution because cross-device communication is not required during each rendering process. In particular, this is suitable for mobile environments, considering that their wireless networks often have a high latency, low bandwidth, and unstable connectivity. Also, FLUID reduces a large amount of network usage and minimizes the number of network round-trips. Therefore, FLUID can provide a better responsiveness than screen mirroring techniques, which renders all device surfaces only on the host device (as discussed in Section XI-C).

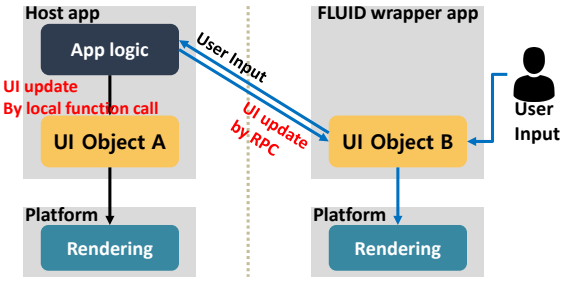



Fig. 4. UI partition and distribution

To enable this design, FLUID aims to identify a minimal-yet-complete set of UI objects and their related UI resources required when rendering the selected UIs, and then distribute only the set to the guest side. For stock Android UIs, it is trivial to find such a set because their related UI resources have already been published as open-source code. However, the same approach cannot be applied to the case of app-specific custom UIs since it is not publicly known which UI resources they use. To address this problem, we utilize static code analysis and runtime object tracking to identify such a set carefully without incurring any false-negative errors. This method allows FLUID to support custom UI elements while maintaining their original look and feel, which is unattainable by the up-to-date methods applying UI distribution [2], [3].

**C2.** FLUID aims to support legacy apps and transparently conduct multi-surface operations by providing the programming abstraction of a single device. When a user interacts with an application through multiple devices, cooperation between the host app logic and guest UI objects should be maintained as if they are running on the same device. In particular, when a user input is provided from a guest surface, the graphical states of the guest UI objects should be updated depending on the host app logic or some functionalities of the host app should be triggered. For instance, if a user touches a video display window when its video clip is paused, the corresponding UI object triggers a proper event listener to handle the touch event. Next, the listener resumes the video clip and synchronizes a progress bar with it. Fundamentally, such cooperation is achieved through local function calls, which are performed within the same address space. Thus, to support extended cooperation between devices, FLUID transforms local function calls into cross-device RPCs in a transparent manner, while solving some technical issues, including 1) function call interception and RPC transmission, and 2) seamless cross-device RPC execution (as described in Section VI). This allows FLUID to support legacy applications without any modification on

their source code, thereby maximizing its applicability.

**C3.** FLUID provides users with another option of replicating some UI elements across multiple devices to provide better flexibility. For instance, a user can replicate a video progress bar on both a smart TV and a smartphone when watching a video image on the former. This replication allows the user to easily control the video clips from both devices. To achieve this, FLUID guarantees that it applies all updates to replicated UIs in a *deterministic* manner to synchronize their graphical states. This implies that FLUID *triggers* and *executes* all updates to the replicated UIs on all devices in the same manner and order. Its details are described in Section VII.

**C4.** FLUID should be able to keep guest UIs up-to-date, even if the device configurations change. When a device changes its configuration, such as screen rotation, screen resizing, and language switching, a mobile app can usually apply a new UI tree at runtime to provide better usability to users. For example, if a smartphone's orientation changes from portrait to landscape, a running host app (e.g., video player) may replace an existing UI tree with a new tree designed for landscape mode. During this process, the app removes all UI objects of the old tree and instantiates new UI objects to construct the new tree. When the UI tree replacement is complete, the UI objects on the guest device become invalid, indicating that the guest UIs should be removed and re-created. Such runtime changes can occur more often than expected. For example, to search for and watch videos on YouTube, users tend to change the orientation of their devices every 3 min on average [13].

To address this, FLUID should seamlessly and efficiently update guest UIs according to runtime changes. Otherwise, the usability of this multi-device system may be severely degraded because a user has to manually redo the UI distribution step for every change in runtime. For seamless runtime change handling, FLUID automatically identifies new UIs visually similar to the guest UIs based on pixel similarity and redistributes them to the guest device. An important technical issue with this approach is to minimize the delay in identification and redistribution for better usability. To this end, FLUID also proposed several optimization techniques.

**C5.** FLUID aims to provide seamless network disruption handling between the host and guest devices to avoid undesirable side effects. The host and guest devices could be unexpectedly disconnected at any time for various reasons (e.g., unreliable wireless signals and battery depletion). Such a situation may lead to severe problems with multi-device usability. For example, the host app may crash because cooperation with a guest UI is not successfully achieved. In this case, users have no choice but to forcibly kill the session of the host app. Moreover, guest UIs may continue to waste resources (e.g., memory and power) despite no communication with the host. To solve this issue, FLUID seamlessly restores the guest UIs on the host device in response to the disconnection, allowing the host app to maintain its execution without any problems. In addition, FLUID properly cleans the residuals of the guest UIs to avoid wasting resources.

## V. Selective UI Distribution

In this section, we describe how FLUID partitions and distributes the UI elements. FLUID provides high responsiveness by deploying a minimal-yet-complete set of UI objects and their relevant resources necessary for rendering. To support this, our system uses static code analysis and runtime object tracking to identify such a set on the host side and applies a cross-device UI re-creation on the guest side.

### A. Static Code Analysis

Before distributing UI elements, FLUID thoroughly analyzes target apps through a static code analysis to identify a candidate set of UI objects and their UI resources that the renderer thread may access. For UI rendering, the renderer accesses the UI resources of each UI object while performing the following rendering functions: *i*) `measure()` to determine the size of each UI element, *ii*) `layout()` to determine the position of each UI element, and *iii*) `draw()` to draw each UI element.

To identify the UI resources of each UI object, FLUID leverages a static analysis technique called the class hierarchy analysis (CHA) through *Soot* [14], an open-source static analysis tool. CHA generates an exhaustive call graph, where each call site refers to all possible class methods that can be invoked. To do so, CHA analyzes all possible class types that each call site object can have according to the class hierarchy relationship (i.e., the declared type of each call site object and each of its children types). However, the current CHA implementation of *Soot* cannot be applied directly to identify UI resources for the following reasons: *i*) it is designed for general programs that have an entry function, such as `main()` (which Android apps do not have), and *ii*) it is designed to analyze an entire program rather than only a specific part.

Thus, we developed FLUID to utilize CHA in three ways. First, FLUID synthesizes a dummy program with a `main()` function, which is used as input for *Soot*. Second, FLUID copies all classes, which are extracted from a target program (i.e., an APK file) and from the platform library (because the platform library defines stock Android UI classes) into the dummy program. Third, FLUID adds a code that invokes the rendering functions of the `View` class (the root UI class in Android) into the `main()` function. Based on these methods, *Soot* can properly execute the CHA starting from `main()` of the dummy program. In addition, it can create an exhaustive call graph that includes all UI objects and their resources that are potentially reachable because all UI classes are derived from View. From the call graph, FLUID can extract a candidate list of all UI classes and UI resources that the target app may use. Notably, because of the nature of CHA, this final list may contain some false-positive errors, but they do not affect the correctness; moreover, no false-negative errors will be found.

The analysis result for the same app does not change unless it is updated. Thus, we can envision an execution model in which a server (e.g., an app market server or a FLUID service server) analyzes each app and puts the analysis result into its APK file whenever its new version is released. We confirmed that the size of the analysis results for 20 legacy apps (described in Section XI-A) ranged from 163 to 236 Kbytes, increasing the size of their original APK files by $0.3\%$ to $13.2\%$. This is an affordable overhead when considering the amount of storage attached to up-to-date smart devices. Furthermore, we confirmed that when our analysis technique is executed on a desktop with 8 cores and 64 GB of RAM, the elapsed time ranges from 119 to 210 seconds for the 20 legacy apps.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2024.3349561



### B. Runtime Object Tracking

FLUID aims to minimize the false-positive errors contained in the analysis results through runtime object tracking. To enable this, we modify a serialization library called *Kryo* [15] according to our goals. Typically, when serializing a target object, the current implementation of *Kryo* explores and serializes all objects reachable from the target object dynamically. Consequently, when FLUID employs *Kryo* without any modification, it may even serialize objects unnecessary for UI rendering (i.e., non-graphical states). Thus, we modify *Kryo* such that FLUID serializes a minimum set of objects (i.e., UI objects and their resources) required to render guest UI elements on a guest device. Considering that the UI objects selected by a user act as input, the modified *Kryo* tracks the UI resources, which are the member fields of each UI object, based on the analysis results described in Section V-A. It can effectively serialize only the intersection between the set of reachable objects that *Kryo* encounters, while exploring at runtime, and the set of reachable objects that our analysis result includes.

### C. Cross-device UI Re-creation

After transmitting the serialized UI resources to a guest device, the FLUID *wrapper* app executing on the guest creates the delivered UI subtree and the associated UI resources using its own UI thread and performs local rendering to show the guest UI elements. We use standard Android APIs to dynamically generate UI elements. Furthermore, an app code and other resource files (e.g., images and fonts) of the host app are necessary to re-create customized UI objects. Accordingly, the *wrapper* app dynamically extracts and loads them from the APK file of the host app passing through the pairing phase.

## VI. Transparent RPC Support

After deploying guest UIs on guest devices, the host app logic and guest UIs must interact with each other. FLUID aims to support the programming abstraction of a single device for multi-surface operations to maximize applicability. Thus, FLUID transparently extends inter-object function calls within the same address space to cross-device RPCs, as shown in Fig. 5(a). This section describes how FLUID supports parent function call interception and seamless RPC execution.

### A. Transparent Function Call Interception

Normally, a function is called by storing its arguments and a return address in registers or the stack and by jumping to the address of the function entry point. On virtual machine (VM)-based systems, such as Android, the VM stores and manages the entry point of each function. Thus, FLUID modifies the Android VM (ART) and intercepts the function calls (see Fig. 5(a)). The entry point address of a target function is replaced with the *code gadget* address of FLUID, which transparently converts local function calls into RPCs (more details are provided in Section X). Upon intercepting a function call, FLUID determines whether it should be handled as an RPC by checking which device the corresponding UI resides in. If the UI resides on the guest side, FLUID creates an RPC message along with the arguments of the target function from

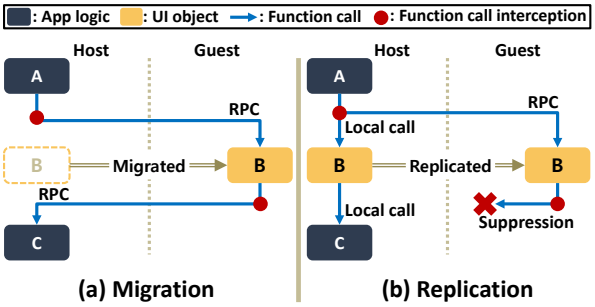



Fig. 5. Transparent RPC support with migrated and replicated UI objects

the registers and stack and transmits it to a guest (callee) device. The FLUID code gadget on the host (caller) device then jumps to the return address of the target function upon receiving a return value or an error code from the callee device. However, if the UI only exists on the host side, FLUID jumps to the original code of the function instead of generating an RPC message. Although an interception can cause an unnecessary performance overhead, in Section XI-B, we show that such overhead is negligible.

### B. Seamless RPC Execution

Even after intercepting local calls and transforming them into RPCs, care must be taken to execute the RPCs properly. Broadly, we address two problems for correctness. Although the discussion in this study considers that a host is making an RPC to a guest, it is equally applicable the other way round. First, a problem occurs when a target function has reference-type arguments, such as uniform resource identifiers, because such references are only valid on the host device. To resolve this, FLUID checks whether each argument is of a reference or value type. If it is the former, FLUID copies the referenced resource object to the guest device and allocates it with a new reference. Then, FLUID on the guest side replaces the reference-type argument with a new reference such that the target function can access the allocated resource correctly on the guest device. Second, when executing a target function, it may access objects that do not reside on the guest device (i.e., non-graphical objects). To enable such accesses, FLUID employs *virtual* objects, which are proxy objects for real objects existing on the host device. When the target function accesses a virtual object, FLUID forwards it to the host device through an RPC (see Fig. 5(a)).

## VII. Cross-device UI Replication

To maximize the flexibility, FLUID provides users with an option to replicate a UI element on multiple surfaces and use it on all or some of the surfaces. For example, multiple users can share the same UI element on their smartphones to perform collaborative tasks, such as playing with a hidden picture puzzle or filling out forms (e.g., inputting passport numbers for a group flight reservation). In this section, we explain how FLUID enables UI replication.

### A. Overview of UI Replication

The basic mechanism for UI replication is the same as the UI distribution and RPC mechanisms described in Sections V and VI, respectively. This means that FLUID deploys the

UI objects and their resources necessary for rendering on guest devices and uses RPCs to transform local method calls into remote method calls whenever graphical states and non-graphical states need to interact across devices. The difference here is that UI replication now displays and manages *replicated* UI elements, as illustrated in Fig. 5(b). Therefore, all devices (host and guest devices) now have a copy of each (replicated) UI element and its graphical states, and synchronize them across all devices. Accordingly, FLUID also allows a user to interact with the replicated UI elements on all or any of the surfaces.

To achieve this, FLUID implements a *dual execution*, where it ensures that the replicated UI states are updated *deterministically* on every device (see Fig. 5(b)). A state of a UI object is updated by executing a UI object method. Using *dual execution*, FLUID invokes a UI object method both on the local device that a user interacts with and on all other devices containing replicas of the UI object. However, the *dual execution* requires us to address two additional challenges, UI state synchronization and duplicate execution suppression, which we discuss in the remainder of this section.

### B. UI Synchronization and its Guarantee

To synchronize the states of replicated UI objects and UI resource objects, FLUID makes deterministic state updates for replicated objects. In other words, FLUID enforces that replicated executions at different devices are identical so that they can produce the same UI states. In general, enforcing deterministic execution requires identifying the sources of non-determinism and enforcing the determinism on them. The sources of non-determinism in a mobile system include thread scheduling, user input, sensor input, network input, file reads, inter-process communication, hardware specification, random numbers, clock readings, and so on. A replication system that enforces determinism must implement a mechanism to make those non-deterministic events deterministic across multiple devices. For example, a previous replication system for mobile devices (Tango [16]) logs all non-deterministic events from a "leader" device and forwards them to a "follower" device. Unlike Tango, which provides full replication, FLUID focuses on UI replication and employs a customized design suitable for UI replication.

The replication design of FLUID leverages the following two observations regarding the UI execution model of Android. First, the Android UI system has only a single UI thread that updates the states of all UI objects. Second, there is a single input event queue that drives the execution of a UI thread. In other words, the execution of a UI thread dequeues an input event from the input queue and executes an event handler associated with the input event. This execution model is not specific to Android. Many UI systems, such as Swing [17], Qt [18], and Cocoa [19], follow this execution model because it avoids race conditions when updating the UI states [20].

Based on these observations, our UI replication forces deterministic updates of the replicated UI states. More specifically, we use two techniques, *deterministic triggering* and *deterministic execution*, to deterministically apply all updates to replicated objects (UI objects and UI resources) in the same manner. The combination of these two techniques guarantees the overall UI state synchronization because every aspect of a UI state update (triggering and executing the update) becomes deterministic.

**Deterministic triggering of UI state updates.** To enforce the deterministic triggering of UI state updates, we ensure that the host device imposes a total ordering of all UI updates, and the guest devices simply follow the ordering from the host. This would be easy if all UI updates were triggered by the events from the host (e.g., user input on the host device), since we would just need to make RPCs to guest devices whenever there is a UI update to trigger the UI update on the guest devices. However, our goal for UI replication is to allow users to interact with replicated UI elements across multiple devices, and the user input on a guest device can potentially change the state of a UI object. Thus, we forward all user input events from the guest devices to the host device, and the host device enqueues the forwarded events to its input event queue to process them. When there is a UI update, the host device executes it locally and makes an RPC to each guest device to trigger the execution of the UI update on the guests. These local and RPC calls occur asynchronously from each other to preserve user interactivity. In addition, we prevent the UI thread running on a guest device (as part of the container app) from updating replicated objects directly. This means that the state of a replicated UI object on a guest device can be updated only by the RPCs from the host.

In summary, the FLUID UI replication processes only two types of input events: (i) all events in the host's input event queue (e.g., user input on the host), and (ii) user input events from guests. All other events from guest devices are excluded from processing because our goal is to enable UI interactions on multiple surfaces. Because the host receives all user input events from guest devices, its input event queue naturally imposes a total ordering of all events that FLUID processes. This mechanism ensures that we trigger the UI state updates deterministically.

**Deterministic execution of UI state updates.** To execute UI state updates deterministically, FLUID relies on the host to supply non-deterministic values while executing a UI state update. For example, consider a method call on a button UI object that updates the state. The execution of the method call might use a hardware-dependent value, such as IMEI. Because different phones will have different IMEI values, we cannot guarantee determinism if we use a local IMEI value while executing this method. Thus, instead of moving objects with non-deterministic values (e.g., hardware information, clock readings, random numbers, or file reads), FLUID creates virtual objects corresponding to them and always receives such values from the host through RPCs. For instance, because most hardware information is accessed by binder objects connected to external system services, our basic mechanism can naturally handle it by simply replacing binder objects with virtual objects.

### C. Duplicate Execution Suppression

As discussed, our UI replication makes the RPCs execute the non-replicated parts of an app. Simultaneously, its original UI residing in the host device also executes the non-replicated parts. Unfortunately, this has an undesirable consequence of a *duplicate execution*, in which a non-replicated method is invoked multiple times owing to replicated executions on

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2024.3349561



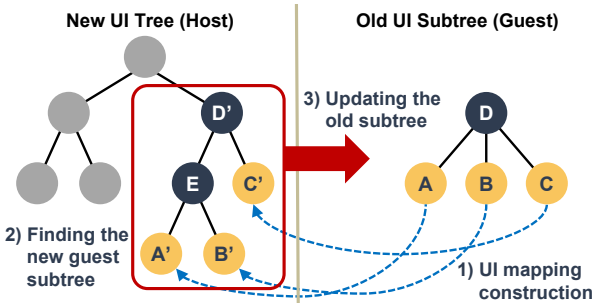



Fig. 6. Cross-device runtime change handling

different devices. For example, suppose a user replicates a video playback control on a guest device from the host. When the user clicks on a play button on either the host or guest surface, FLUID arranges each device to update the graphical state of the play button through a *dual execution*. Each device then makes an individual RPC to the host app logic (which is a non-replicated part) to start the video. However, this will result in two calls, and the host app logic will end up toggling the video status twice, which will pause the video instead of playing it. To address this problem, FLUID caches the result of a method execution and provides the cached result to an RPC (see Fig. 5(b)).

## VIII. Cross-device Runtime Change Handling

Once some device configurations (e.g., screen orientation) are changed, a host app may replace its current UI tree with a new one. In this case, FLUID aims to seamlessly replace the UI subtree on the guest device without any user intervention. To this end, FLUID automatically identifies new UIs visually similar to the old guest UIs and redistributes them to the guest side. With this approach, the delay must be minimized because the UI distribution should be repeated whenever runtime changes occur. This may cause significant inconvenience to users, in contrast to the first UI distribution, which is a one-shot cost to initiate a multi-surface interaction. This section presents how to *i*) automatically identify new UI objects to be redistributed, *ii*) minimize the UI identification overhead, and *iii*) minimize the UI redistribution overhead.

### A. UI Mapping Construction

In general, when device configurations are changed, mobile apps typically tend to keep the overall look of their screen as before. For instance, a video player (e.g., a VLC player) shows common buttons for video control, such as *play* or *forward*, in both portrait and landscape UI trees. Based on this intuition, FLUID constructs mappings between the widgets of UI trees to identify new UI elements that are similar to old guest elements. A widget is a special UI object with graphical content, from which we can extract RGBA pixels. As shown in Fig. 6, FLUID calculates the pixel similarity between widgets residing in the different UI trees (i.e., the old subtree on the guest side and the new tree on the host side) based on the Euclidean distance [21], [22], and identifies pairs of widgets that have the highest similarity (i.e., UI mapping). If an old UI widget cannot find proper UI mapping, FLUID asks the user to reselect UI elements to distribute to the guest side.

### B. UI Mapping Optimization

Although similarity-based UI mapping enables seamless runtime change handling, the application of this method may result in substantial delays. The first is a network delay for transmitting the old guest widget pixel data to the host side for the similarity calculation. To alleviate this problem, FLUID precaches the latest graphical states of the guest widgets to the host device when they are refreshed by the RPCs. This method allows FLUID to calculate the pixel similarity without transferring all pixel data because it can extract the pixels of the guest widgets on the host side through rendering with the precached data. FLUID precaches graphical states instead of pixel data because such states contain rich information and have a small size. In addition, the graphical states can enable recovery from network faults (as described in Section IX). Keeping the cache up to date can cause significant overhead because it may require frequent network transmissions. However, precaching can run seamlessly in the background after the guest UI changes. This allows FLUID to hide network delays from users. Note that for UI replication, precaching is not required because copies of the guest widgets already exist in the host device and FLUID can extract pixels from them.

Another delay factor is the number of pixel similarity calculations that increase with increase in the combination of UI pairs. To alleviate such overhead, FLUID eliminates unnecessary calculations by matching the identifiers of the widgets. App developers often assign the same identifiers to UI objects, even in different UI trees, as long as they are used for the same function. Such a programming convention allows FLUID to create UI mapping between widgets without calculating the pixel similarity; if some widgets have the same identifiers in both old and new trees, then FLUID can immediately make mapping between them.

### C. Cross-device UI Partial Update

After constructing UI mappings, FLUID finds the subtree including the new target widgets and transmits it to the guest side through the same UI distribution method described in Section V. This step inherently entails non-negligible network delays, which may cause significant inconvenience to users. To mitigate such delays, FLUID transmits only the differences in graphical states between the old and new subtrees, instead of all data. Typically, UI widgets with a high visual similarity are likely to have similar graphical states. Based on this intuition, FLUID updates the old widgets placed on the guest device identically to the new ones by applying only the differences in graphical states between them. FLUID can also update all layout objects in the same way as long as the new subtree has the same hierarchy as the old one. The partial UI update works as follows. Upon receiving graphical states from the host device, the FLUID wrapper app creates a new guest subtree to replace the old one. To do so, the wrapper app first updates the old UI objects to make them identical to the new ones or creates newly added UI objects. Subsequently, it constructs a new guest subtree by linking these objects and displays them through local rendering.

## IX. Seamless Network Fault Handling

Typically, traditional mobile systems require app developers to directly implement app logic for handling network faults.

In other words, when the connection to an external device is abruptly lost, a mobile app responds by executing pre-written exception handlers. However, this app-level approach is not suitable for FLUID for the following two reasons: 1) The multi-surface interaction of FLUID is transparent to the apps; this indicates that a network fault during multi-surface interaction cannot trigger the mobile app's exception handlers properly, leading to unexpected app crashes. 2) Most apps respond to network faults in an explicit manner, such as by displaying an error message or asking a user to reconnect to the remote device. This can significantly disrupt the user experience for multi-surface interaction because the continuity of app usage is broken. For these reasons, FLUID has a new system-level mechanism to handle network faults seamlessly, without intrusive impact on app behavior.

Our system handles unexpected network faults through two steps: fault detection and seamless fault recovery. For fault detection, FLUID employs the heartbeat mechanism, which is widely used in the network domain [23]. FLUID periodically sends heartbeat messages to the guest *wrapper* app with a regular interval, and then the *wrapper* app responds immediately upon receiving the heartbeats. If the host side does not receive any responses within a timeout threshold, or if the *wrapper* app does not receive any heartbeat messages at regular intervals, they consider it a dysfunction. In the current FLUID prototype, we empirically set both the heartbeat interval and time-out threshold to 100 ms.

For fault recovery, FLUID seamlessly restores all the guest UIs to the host device, enabling the app to continue the execution with the restored UIs. To this end, we should address the following two issues. First, we need to bring the up-to-date graphical states of all guest UIs to the host device before the network fault; however, this is challenging because it is impossible to predict when a network fault will occur. To address this, FLUID uses the graphical states of the guest UIs precached for pixel similarity calculation as checkpoint data. When a network fault occurs, FLUID can restore the guest UIs appropriately on the host device via the latest checkpoint. Since the precaching is performed every time each RPC function updates the guest UIs (as described in Section VIII-B), this checkpoint-restore approach ensures that the restored UIs have the same graphical state as the guest UIs before the network fault. Second, when a network fault occurs during RPC execution, FLUID should be able to roll back the execution context of the host app to the state before the RPC function was invoked. And then, it should invoke the original local function corresponding to the RPC function to continue the execution of the host app. To this end, FLUID halts the host app from waiting for the RPC function to return and restores its registers and call stack to the state prior to the RPC invocation. FLUID then sets the value of the program counter to the entry address of the corresponding local function to invoke it. This allows the host app to continue its execution seamlessly without the app crashing or interrupting.

## X. Implementation

We implemented a working prototype of FLUID by adding new system components to Android or extending its existing system components. In particular, we modified the *Kryo* library to serialize and restore UI objects and UI resources based on static analysis results while adding it as a system feature to Android (2,399 LoC). We have extended the Android Java framework to manage distributed UI subtrees and event queues of UI threads across multiple devices (1,766 LoC). Moreover, we added a special code gadget that consists of C++ native code and ARM assembly code to Android VM (i.e., ART) to transparently hijack the control flows (1,874 LoC). We also added a system service to manage network connections between devices (1,687 LoC). This section focuses on implementation details only for the special code gadget added to ART, which is one of the main techniques to enable a transparent cross-device RPC and seamless dual execution.

**Function call interception.** To transparently intercept local function calls, as described in Section VI, we significantly revised the Java class loader of Android ART. When a Java class is loaded into memory, the class loader of Android ART generates `ArtMethod` objects that manage the metadata of each function belonging to the class. The class loader then stores the entry point of each function in each `ArtMethod` object. At this point, to capture invocations to each function later, we configure the entry point of each function to the address of the FLUID code gadget in advance.

**Function return interception.** As mentioned in Section VII-C, FLUID uses the link register (LR) that stores the return address of the function to record a result value returned from a function. After intercepting a call to the target function, FLUID stores the address of its code gadget in the LR and jumps to the original entry point of the target function. After completing the execution of the function, the control flow is returned to the code gadget address stored in the LR. FLUID can then successfully intercept the return value passed by the target function.

**Conflict with garbage collector.** The primary issue encountered when implementing the FLUID code gadget is conflicting with a garbage collector (GC) executed in Android ART. When a call to the target function is intercepted, the assembly part of the code gadget first stores the context information (e.g., general registers) of its caller on the stack. Then, the code gadget performs computation to determine certain aspects, such as where the call should be forwarded and whether the call is related to a replicated UI object. Java VMs (e.g., Dalvik) typically manage two separate stacks, one for the Java region and the other for the native region, whereas Android ART maintains a unified stack for those two regions. This unique characteristic of ART may induce conflicts with the FLUID code gadget when the GC walks through the stack to check Java objects in the heap. Specifically, unexpected errors may occur because the GC cannot parse some stack frames of the native region, which are created by FLUID to save context information. To solve these error situations, we modified the ART GC such that it can identify and skip the stack frames generated by the FLUID code gadget. This allows the GC to walk through the Java stack frames without errors.

## XI. Evaluation

We implemented a working prototype of FLUID to demonstrate and evaluate its functionality, i.e., selectively distributing the UIs of legacy apps to multiple devices and supporting cross-device cooperation between the distributed UIs and app logic transparently.

| Type | User case scenario | Custom UI type | App name (Downloads) | FLUID | | | | App migration size (Kbytes) |
|---|---|---|---|---|---|---|---|---|
| | | | | Network round-trip | Arg. (bytes) | Ret. (bytes) | UI distribution size (Kbytes) | |
| Usability | Edit text on different device | Edit Text | Color note (100M) | 1 | 0 | 0 | 8.7 | 16,500 |
| | | | Text editor (1M) | 1 | 587 | 0 | 35.8 | 46,100 |
| | Control media with different device | SeekBar, Button | VLC Player (100M) | 1 | 14 | 0 | 998.5 | 38,000 |
| | | | Music Player (0.5M) | 1 | 128 | 0 | 358.4 | 18,700 |
| | Control painting tool with different device | Scroll, Button, Image | PaperDraw (10M) | 1 | 666 | 0 | 2,026.1 | 21,300 |
| | | | Paint (1M) | 1 | 1,602 | 0 | 272.5 | 63,400 |
| | Chatting with different device while broadcasting | Edit Text, Button | LiveMe (50M) | 2 | 72 | 1 | 45.3 | 85,600 |
| | | | Afreeca TV (10M) | 5 | 8 | 1 | 234.3 | 43,000 |
| | Search destination with different device | Edit Text, Button | Naver map (10M) | 1 | 67 | 1 | 37.9 | 199,000 |
| | | | Maps.me (50M) | 1 | 0 | 1 | 126.1 | 94,500 |
| | Read document with different device | Text, Scroll | File Viewer (1M) | 0 | 0 | 0 | 8.9 | 11,700 |
| | | | Bible KJV (10M) | 0 | 0 | 0 | 20.7 | 97,200 |
| Privacy | Login with personal device | Edit Text | Instagram (1B) | 1 | 5 | 1 | 12.5 | 45,900 |
| | | | PayPal (50M) | 1 | 94 | 0 | 19.6 | 54,000 |
| | Unlock pattern with personal device | Pattern | Smart AppLock (10M) | 1 | 8 | 0 | 3.6 | 29,600 |
| | | | AppLock (10M) | 1 | 8 | 0 | 106.3 | 67,500 |
| | Sharing photo to public device | Image | Gallery (10M) | 0 | 0 | 0 | 182.9 | 51,200 |
| | | | A+ Gallery (10M) | 0 | 0 | 0 | 362.8 | 66,300 |
| Collabo. Use | Fill in information collaboratively | Text, Edit Text | eBay (100M) | 1 | 55 | 1 | 62.9 | 73,500 |
| | | | Booking.com (100M) | 1 | 67 | 1 | 97.7 | 93,000 |

TABLE I
USE CASE LIST FOR COVERAGE TEST. "CUSTOM UI TYPE" IS A SUPER CLASS OF DISTRIBUTED UI. "ARG." AND "RET." INDICATE THE MAXIMUM AMOUNTS OF DATA TRANSFERRED FOR THE ARGUMENTS AND RETURN VALUES OF RPCS, RESPECTIVELY.

The FLUID prototype was implemented based on the Oreo version (v.8.1.0) of the Android Open Source Project (AOSP) and ported to two types of devices: a Google Pixel XL (smartphone) and Pixel C (tablet). During our evaluation, we set all cores of the experimental devices to operate at the maximum CPU frequency ($2 \times 2.15$GHz & $2 \times 1.6$GHz). In addition, all devices were connected to the same Wi-Fi access point, which has a throughput of 45 Mbps and a round-trip time with a median, average, and standard deviation of 32.1, 42.9, 40.31 ms, respectively.

### A. Coverage Test

We first explored how well FLUID can support unmodified legacy applications using the 10 use-case scenarios described in Section II. We used two legacy apps downloaded from the Google Play store for each use-case scenario. Table I lists the use cases and legacy apps used in the coverage test. We observed that all 20 legacy apps have their own custom UI elements, and FLUID supports them for a successful display on multiple surfaces of heterogeneous devices: phone-to-phone, tablet-to-phone, and phone-to-tablet.

The "network round-trip" column in Table I indicates the maximum number of data transfers between the host and guest devices while executing a single method on the guest device in each use case. It is notable that the number of cross-device communication ranges from only 0 to 5, and is mostly 1; in addition, no communication occurred for the use cases without a user input (e.g., Gallery and File Viewer). This means that FLUID can minimize the number of network round-trips because its selective UI distribution method has successfully divided a complete set of UI objects and their resources (as described in Section V).

The "UI distribution size" column indicates the amount of data transferred by FLUID for UI migration or replication. The "app migration size" column indicates the amount of data in memory (i.e., code, data, stack, and heap) allocated by each app immediately after launching it (i.e., when its memory usage is at its least). Note that we measured the data using the Android Profiler tool [24]. Because app migration techniques, such as Flux [6], transfer the entire memory of an application to another device, this measurement can serve as a lower bound on the amount of data transfers when we attempt to utilize the app migration techniques. Consequently, we confirmed that the data transfer size for app migration is incomparably larger than that of the UI distribution. This means that FLUID transmits an extremely small portion of the total app memory by identifying a minimal-yet-complete set of UI resources to support an efficient UI distribution.

### B. Performance Test

We quantitatively evaluate the performance of FLUID for its multi-surface operations. We repeated each experiment ten times with each configuration.

**UI distribution time.** We evaluated the performance of FLUID in placing UI elements on a guest device by measuring the *UI distribution time*, which is defined as the elapsed time from when a user triggers the UI distribution to when the last screen update on the guest side is completed. Therefore, it includes the time required for network transmission and screen rendering. Fig. 7 shows the breakdown of the UI distribution time measured for the 20 legacy apps. The distribution time ranges from 132 to 735 ms according to the types of UI elements distributed from each app because each UI element is associated with different sets of UI objects and their resources

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2024.3349561



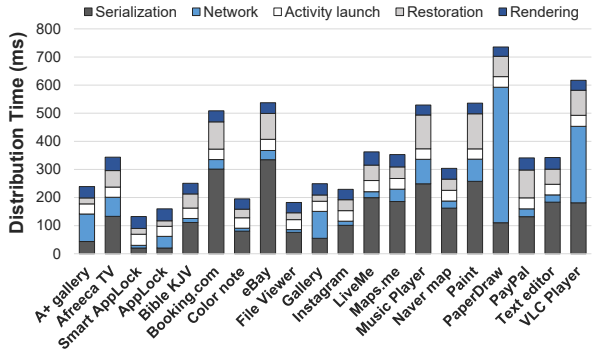



Fig. 7. UI distribution time of FLUID

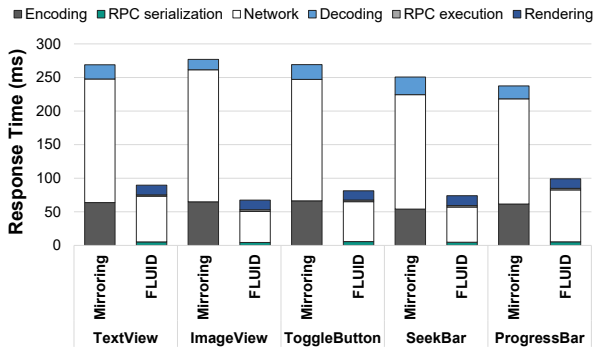



Fig. 8. UI response time of FLUID

required for rendering. The result shows that FLUID distributes UI elements to another device sufficiently fast to support interactive use.

Notably, the serialization process incurs a large overhead in many cases. As described in Section V, we modified the *Kryo* serialization library [15] in our FLUID working prototype. Although our modification minimizes the number of data on serialized objects, the serialization overhead remains high. We observed that there are some opportunities to optimize the performance of *Kryo*, such as removing deep recursions included in its original implementation, which likely can significantly reduce the performance overhead. Alternatively, we expect that using different serialization libraries that are more optimized for performance can also solve this issue.

**UI response time.** To evaluate the responsiveness of UIs distributed on a guest device, we measured the *UI response time*, which is defined as the elapsed time from when a user touch input is provided on the guest surface to when the result of the touch input is displayed on the guest surface. Fig. 8 shows the average UI response time taken by FLUID to update the five most popular UI widgets compared to an open-source screen mirroring program, SCRCPY [11]. The main factor that affects the UI response time of FLUID is the size of the arguments passed to an RPC, which was set to 2,000 bytes in this experiment. Note that the size is larger than the largest argument size (1,602 bytes) observed in the 20 legacy apps (See Table I). Our results show that FLUID has a 2–4-times faster responsiveness than the screen mirroring method.

**RPC performance overhead.** To explore the overhead that FLUID imposes while supporting the cross-device RPC described in Section VI-A, we compared the function execution times for three cases: *i)* without any interception (i.e., on a stock Android), *ii)* with interceptions but no RPCs, and *iii)* with both interceptions and RPCs. In this experiment, we used a custom app that invokes a simple function `TextView.setText()` 100 times. Our results show that, among the three cases, the average execution times for the function were 215, 221, and 2,811 µs (with standard deviations of 63, 61, and 876 µs, respectively). This indicates that FLUID only incurs an affordable overhead, 6 µs on average, for intercepting a function call. By contrast, the third case incurs a relatively large latency because of the RPC message transmission. Although this is an inevitable cost required for cross-device communication, we expect that this issue can be mitigated through advanced wireless technologies such as 802.11ad.

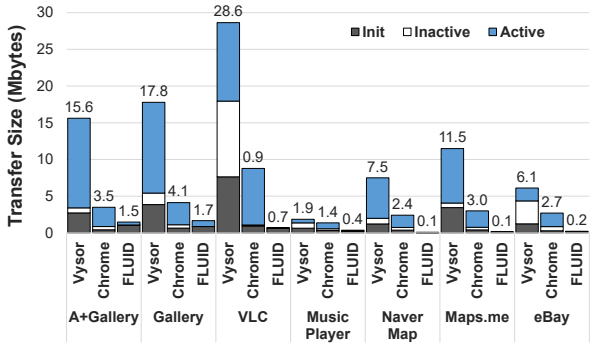



Fig. 9. Data usage comparison

## C. Network Optimization

As shown in Fig. 8, a network transfer occupies a large part of the UI response time for both FLUID and SCRCPY. To further explore the impact of network usage, we measured the amount of network data transmitted in the following three stages: *i) init*: it denotes the initialization for user interaction, i.e., launching an app for screen sharing and deploying some UI elements to a guest device for FLUID. *ii) inactive*: an idle period, without interaction, that lasts until the 10 sec mark (i.e., no user input is provided), and *iii) active*: an active period from starting touch events, after the 10 sec mark. During this experiment, we compared the network usage of FLUID and screen mirroring approaches utilized for various multi-surface use cases. FLUID moves only those UI elements selected by users to different devices for the same use cases, whereas screen mirroring duplicates the entire screen of one device to another. The differences in these approaches lead to different patterns of network usage, which can affect both the response time and energy consumption.

We conducted this experiment using the seven legacy applications shown in Table I, and we used Chromecast and Vysor to experiment with screen mirroring approaches. Fig. 9 shows the total amount of data transmitted in each application. We confirmed that FLUID uses a considerably smaller traffic than the screen mirroring methods in all cases. Vysor provides screen sharing with the full resolution of the host, whereas Chromecast supports a lower resolution to decrease the network usage. Notably, FLUID can support the same resolution as Vysor with a much smaller amount of network usage than Chromecast.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2024.3349561



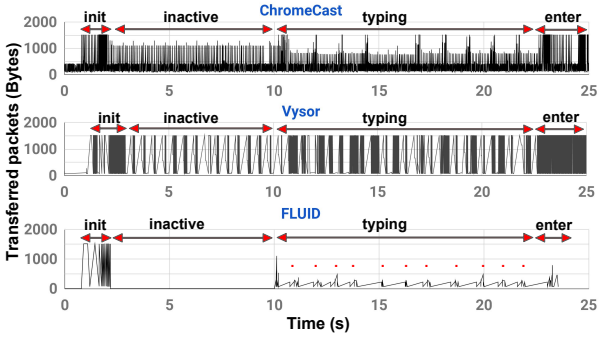



Fig. 10. Byte transfer over time

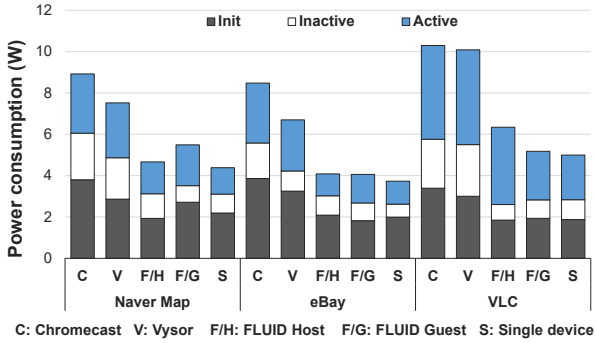



Fig. 11. Power consumption comparison

We further investigated the differences in the network usage patterns of these approaches by focusing on Naver Maps, which is a widely used map app. The Naver map scenario involves moving a query input box for location searching to a guest device, setting the focus to the input box by tapping, typing a location name with 11 characters, and tapping the enter key in the input box. Fig. 10 shows the amount of network transfer over time divided into three stages: *init*, *inactive*, and *active*. The time intervals in which network transfer occurs frequently are represented by high-density areas. Vysor has different densities over time because it uses an adaptive frequency for screen updates. In the active stage, Vysor has denser lines than in the inactive stage because it updates the guest screen more frequently when giving a user input. Consequently, it transmits a significant number of data. Meanwhile, Chromecast has similar densities that appear periodically because it contains transmitted frames at a static frequency, regardless of the user input. Thus, Chromecast uses a smaller number of network data than Vysor, instead of supporting lower resolution mirroring. By contrast, FLUID has a completely different pattern of network usage. It has transmitted a near-zero amount of network data in the inactive stage. In the active stage, it has transferred a much smaller amount of network data at a much lower frequency than the mirroring approaches. This is because FLUID transmits only the data (e.g., RPC arguments and return values) required for cross-device cooperation through RPCs. Notably, there are 11 peaks (marked with red dots) between focusing on the input box and tapping the enter key. These peaks match accurately with the number of characters in the location name used as a query keyword during this experiment. This result indicates that FLUID optimizes the network performance by reducing both the numbers of data transfers and network round trips.

### D. Power Consumption

We measured the power consumption of FLUID through a Monsoon Power Monitor [25] while running three legacy apps under the same scenario used for Fig. 9. Specifically, we measured the average power consumed in five different device cases, i.e., the host and guest devices of FLUID, the host devices of Chromecast and Vysor, and a single device system. As shown in Fig. 11, FLUID consumes power comparable to the single-device case for both the host and guest devices. Chromecast and Vysor have considerable power overhead compared to FLUID.

To further investigate the cause of the power consumption, we plotted the amount of power consumed by Naver Maps

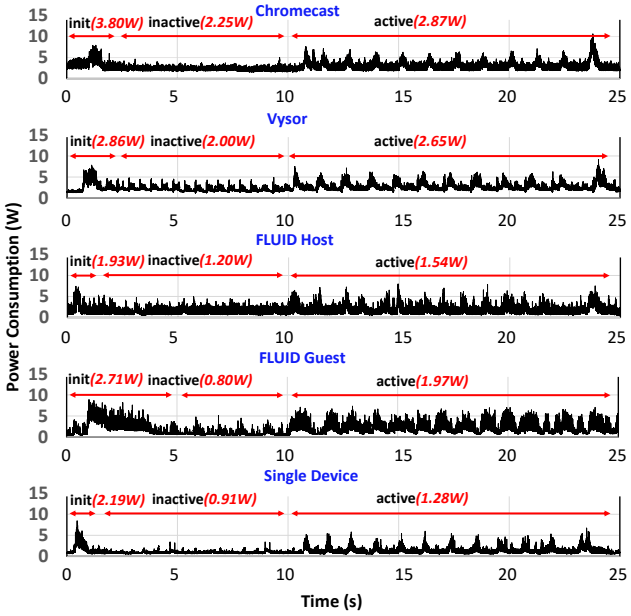



Fig. 12. Power consumption over time

over time, as shown in Fig. 12. We confirmed that network usage primarily causes the power consumption overheads of FLUID, Chromecast, and Vysor, which follows the same trend as a byte transfer over time (Fig. 10).

We did not explicitly solve power optimization in this study. Instead, we focused on optimizing the performance by minimizing the network latency, as evaluated in Section XI-C. Notably, the efficient use of wireless networks reduces the energy consumption because network usage dominates the power consumption overhead.

### E. Runtime Change Handling

**The overall performance.** We quantitatively evaluated the ability of FLUID to handle the changes in runtime. Among the 20 legacy apps listed in Table I, we selected some apps that apply new UI trees when changing the device configurations (e.g., device orientation changes, screen resizing, and language switching). We measured the average elapsed times of updating the guest UI objects when applying new UI trees to the host device. We also compared the performances of the two approaches: *BASE* and *OPT*. *BASE* is a baseline approach in which no optimization techniques are applied. It brings all necessary data from the guest device on demand and then

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2024.3349561



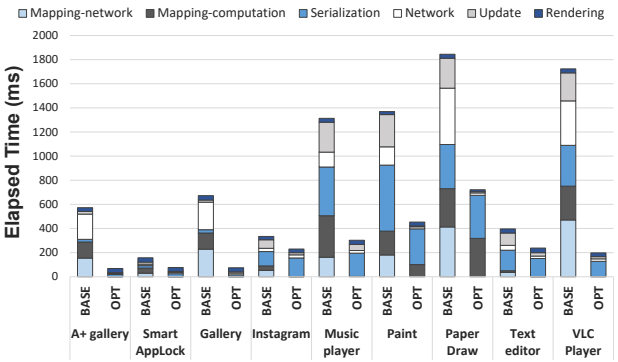



Fig. 13. Runtime change handling time for legacy apps

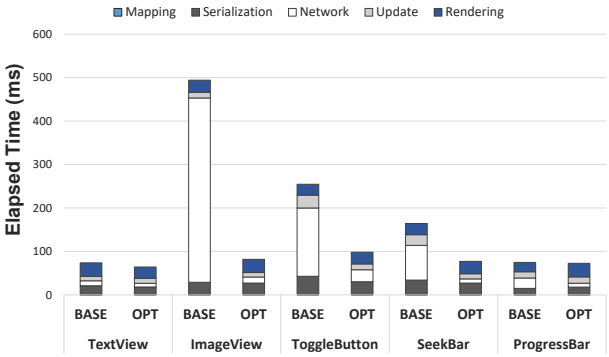



Fig. 14. Runtime change handling time for UI types

constructs UI mappings in a brute-force manner by calculating the pixel similarity for all combinations of the UI widgets. Based on the mappings, it redistributes new UI objects to the guest device by transmitting the entire data. By contrast, *OPT* utilizes our optimization methods for constructing UI mappings and updating the guest UI objects, as described in Section VIII-A.

Fig. 13 depicts the elapsed times in handling the changes in runtime and shows that *OPT* can handle runtime changes by 1.5–9-times faster than *BASE*. This means that *OPT* can effectively reduce the elapsed time required to construct UI mapping information and the amount of data required to update the guest UI objects. Regarding UI mapping construction, we observed that *BASE* has considerable overhead in bringing all graphical states of the guest UI widgets to the host side (corresponding to *Mapping-network*) and in calculating the pixel similarity for all combinations of widgets (corresponding to *Mapping-computation*). However, *OPT* generates no network overhead during mapping construction because it precaches the up-to-date graphical states of the guest UIs whenever they are refreshed on the guest side. In addition, we observed that most legacy apps assign the same identifiers to UI objects used for the same function. This allows the *OPT* to avoid unnecessary computations for pixel similarity by matching their identifiers.

Our results show that only 0.83 ms on average is required to compute the pixel similarity between widgets of different trees. As an exceptional case, the mapping-computation time of *PaperDraw* is comparable to that of *BASE*. This is because many widgets were defined without identifiers, despite the same functions. In addition, *OPT* reduces the network transmission times by 68.2% on average because it uses lesser data than *BASE* to update the old guest UIs into new UIs. Notably, *BASE* transmits an amount of data similar to the UI distribution sizes specified in Table I, whereas *OPT* transmits only 0.1% to 34.3% of the data amount.

We further examined how much *OPT* can reduce the UI distribution time depending on the type of guest UI. Fig. 14 shows the average completion times in handling runtime changes for the five most popular UI widgets with *BASE* and *OPT*. According to the results, *OPT* outperforms *BASE* by 2–6-times in the cases of UI widgets that handle large amounts of graphical states (i.e., ImageView, ToggleButton, and SeekBar). However, *OPT* and *BASE* exhibit a similar performance for specific widgets, such as TextView and ProgressBar, because of their relatively small amounts of graphical states. Additionally, both *BASE* and *OPT* have an insignificant mapping overhead because we have used custom apps that have only one UI object of each type under the replication mode of FLUID.

| Size (w × h) / # of widgets | 100 × 100 | 200 × 200 | 400 × 400 | 800 × 800 |
|---|---|---|---|---|
| 2 | 2.31 | 5.23 | 17.04 | 63.74 |
| 4 | 5.80 | 15.09 | 53.16 | 199.42 |
| 8 | 15.19 | 49.03 | 179.93 | 686.66 |
| 16 | 50.21 | 178.76 | 658.10 | 2563.65 |
| 32 | 185.16 | 648.96 | 2549.64 | 9954.34 |

TABLE II
ELAPSED TIMES FOR UI MAPPING CONSTRUCTION (MS)

**UI mapping overhead.** To further explore the overhead in constructing UI mappings, we measured the execution times when calculating the pixel similarity with varying numbers and sizes of UI elements. To this end, we used custom apps comprising several image widgets, with various colors and without any identifier, and moved all of those widgets to the guest device. The app changes the position of each widget by applying a new UI tree whenever the device orientation changes. Table II presents that the execution time of the pixel similarity calculation increases as the number and size of the widgets increase. This indicates that FLUID incurs an affordable overhead for typical cases. Furthermore, FLUID can create a long delay (approximately 10 s) in an extreme case. However, in practice, most mobile devices have limited screen resolutions (e.g., 2,732 × 2,048 for an Apple iPad pro [26]); thus, mobile apps usually do not encounter such extreme cases. For instance, we observe that legacy apps listed in Table I only have 10 visible widgets on average (maximum of 39), and only 4% of them have a size of 800 × 800 or greater.

**UI mapping accuracy.** To further investigate the accuracy of UI mapping, we conducted extensive experiments using 50 legacy apps with a high number of cumulative downloads from the Google Play Store. Note that we excluded 3D games and web applications that FLUID cannot support, as described in Section XIV. During these experiments, we changed the device configuration (e.g., display size) on the screen that each app first shows after launching. If the app generated a new UI tree for the changed configuration, FLUID tries to find widget pairs from the old and new UI trees based on pixel similarity. Note that a correct widget pair consists of one widget from the old tree and one from the new tree, both of which have identical

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2024.3349561



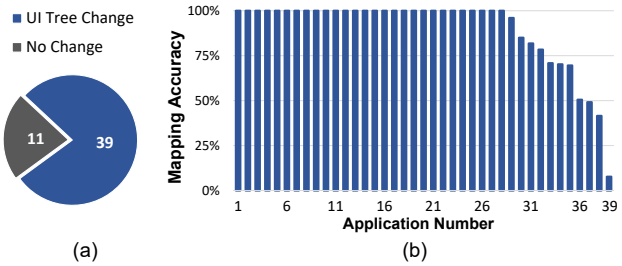



Fig. 15. (a) The ratio of UI tree change on runtime change for 50 legacy apps. (b) UI mapping accuracy for 39 individual apps with UI tree change.

appearance and functionality. We also identified the ground truth of correct widget pairs using Android Studio's layout inspector tool. After that, we evaluate the mapping accuracy, which is defined as the ratio of the number of correct widget pairs over the total number of widget pairs.

Fig. 15(a) first shows how many of the 50 legacy apps change their UI trees when the device configuration is changed. We observed that 11 out of them utilize the existing UI tree without changes to a new one, presumably to avoid the extra overhead of re-rending the screen. On the other hand, the remaining 39 apps (78% of the total) create new UI trees, which potentially lead to usability degradation if FLUID distributes some UIs to guest devices.

Then, we measured the mapping accuracy of FLUID for the 39 legacy apps which create new UI trees. Fig. 15(b) illustrates that FLUID can find all widget pairs completely identical to the ground truth for 28 legacy apps (71.8% of the total). This is because these apps maintain the same visual appearance with the same set of widgets as before, even if the UI tree is changed due to device configuration changes. On the other hand, for the 10 legacy apps (with app numbers 29–38), the mapping accuracy drops up to 41.5% because the properties of some widgets are significantly changed even though the old and new UI trees entirely show similar visual appearances. More specifically, we observed that the resolution of some text/image widgets varies greatly in the new UI tree, even if they show the same content. The large resolution differences between these widgets can significantly reduce the accuracy of our Euclidean distance-based pixel similarity estimation. This could be mitigated if FLUID employs advanced algorithms such as histogram or keypoint matching. Meanwhile, in the 39-th app (i.e., Google Calendar), the new UI tree is rendered to a completely different screen from the old UI tree, resulting in an accuracy drop to 7.7%. In such a case, FLUID could not handle runtime changes seamlessly, thus it should ask a user to re-distribute UIs to resume the multi-device interaction. However, this is a rare case that we only found in one of the 50 legacy apps used in our experiments.

Furthermore, we investigated how many similar graphical states exist between widget pairs. To do this, we extracted graphical states from the correct widget pairs and measured the similarity between two widgets in each pair. Note that the similarity of graphical states is defined as the ratio of the number of states of a new widget identical to those of an old one over the total number of states of the new widget. Table III shows that the graphical states of each widget pair are identical over 92.1% on average regardless of widget types. And we confirmed the data size of non-identical graphical states between each widget pair is less than 700 bytes on average. This implies that when a UI tree is changed due to runtime changes, FLUID can update remote UIs successfully by transmitting only the differences in graphical states between the old and new widgets, which can be a good chance for reducing network overhead greatly as shown in Fig. 14.

| Widget Type | Avg. similarity in graphical states | Avg. difference in graphical states (bytes) |
|---|---|---|
| TextView | 94.6% | 89 |
| ImageView | 97.3% | 384 |
| Button | 96.5% | 193 |
| Checkbox | 99.8% | 110 |
| EditText | 92.1% | 476 |
| ProgressBar | 96.2% | 677 |
| SurfaceView | 93.0% | 114 |

TABLE III
THE AVERAGE SIMILARITY AND DIFFERENCE IN GRAPHICAL STATES

| | v.6.0 to v.8.1 | v.8.1 to v.6.0 | v.7.0 to v.8.1 | v.8.1 to v.7.0 |
|---|---|---|---|---|
| Same Field | 96% | 87% | 98% | 94% |
| Same Method | 90% | 82% | 96% | 92% |

TABLE IV
COMPATIBILITY ACROSS DIFFERENT ANDROID VERSIONS

### F. Compatibility

The core techniques of FLUID, including UI serialization and cross-device RPC, depend on Android's class definitions. However, each class can be defined in different forms according to the version of Android, and FLUID might not operate properly between devices with different Android versions. To check the variance of class definitions, we measured the differences in methods and fields that common classes have between *Oreo* (v.8.1), *Nougat* (v.7.0), and *Marshmallow* (v.6.0). Table IV shows that most of the methods and fields are the same for these versions. Therefore, we can infer that many Android UI classes include common fields and methods as a base.

To confirm our hypothesis, we added a simple translation layer to the UI serialization and cross-device RPC mechanisms to translate four types of UI objects from *Oreo* to *Marshmallow*: `TextView`, `Button`, `EditText`, and a custom UI (`SeekBar`). The prototype successfully serializes common fields while adding dummy data for other fields and translates common methods of the four types of UI objects (e.g., *setText()*), following the class definitions of the *Marshmallow* version. We observed that all four UIs and their common methods operate without error and retain the same look and feel across *Oreo* and *Marshmallow*. We leave automating this translation layer to future study.

## XII. USABILITY STUDY

We further evaluated the efficacy of FLUID through usability studies that respectively assess the perceived performance, intuitiveness, usefulness, and satisfaction of mobile users.

**Participants and Study Procedure.** Through an online community posting, we recruited 15 individuals (10 males, 5 females, mean age 24.7) to participate in our usability study. During each study session, we asked them to perform four

| Application | Performance | | Intuitiveness of UI selection [0,6] | Usefulness [-3,3] | Satisfaction [0,6] |
|---|---|---|---|---|---|
| | Distribution [0,6] | Response [0,6] | | | |
| LiveMe | 5.33 | 5.40 | 4.47 | 2.13 | 4.73 |
| PaperDraw | 4.53 | 5.87 | 3.93 | 2.47 | 5.33 |
| Naver map | 5.40 | 5.53 | 5.20 | 2.27 | 4.80 |
| VLC Player | 4.26 | 5.33 | 4.13 | 2.33 | 5.07 |

TABLE V
USER STUDY RESULTS ACROSS FOUR LEGACY APPS

apps in sequence, with single- and multi-device environments. And then, participants were asked to provide ratings on Likert scales about various aspects as follows:

- Performance for UI distribution/response: How much delay did you feel when distributing UIs to the remote device and touching the distributed UIs?
- Intuitiveness of UI selection: How intuitive do you think the UI selection interface of FLUID is?
- Usefulness: How useful do you think the multi-device interaction is, compared to the single-device environment?
- Satisfaction: What is your overall satisfaction with using a legacy app on FLUID?

**Study Scenarios.** To evaluate FLUID on diverse user scenarios, we thoughtfully selected four legacy apps and gave a short task scenario for each of them as follows.

- Live broadcast (LiveMe): Chat with other viewers while watching the live broadcast video on the full screen.
- Painting (PaperDraw): Draw an animal using at least three different painting tools/colors.
- Navigation (Naver map): Enter a destination on the search input box for route navigation.
- Video player (VLC Player): Find specific scenes using several control UIs while watching a video clip.

**Results and Findings.** Overall, participants responded with positive feedback. As shown in Table V, all apps result in positive scores, in particular at least 4.26, 4.13, and 4.80 for the perceived performance, the UI selection intuitiveness, and the overall satisfaction, respectively (0: *very bad*, 6: *very good*). Furthermore, the usefulness scores were also positive for all apps (-3: *not useful at all*, 3: *very useful*—compared to using a single device).

One interesting finding from the study is that PaperDraw and VLC Player received relatively low scores from participants for the UI distribution performance and the intuitiveness of UI selection. This is due to the fact that both apps require participants to select and distribute a lot of UI elements (i.e., 7–20 widgets). In our post-survey, the participants provided feedback regarding the reasons for the relatively low scores for the two apps. Firstly, some participants reported experiencing a perceptible delay when distributing many UIs to the remote device. However, they noted that this one-shot overhead did not significantly affect the overall satisfaction or usability of the apps since FLUID provided fast responsiveness to the distributed UIs after the initial delay. Secondly, some participants found the UI selection interface of FLUID somewhat cumbersome and unintuitive as it required users to select multiple UIs one by one. They suggested that it would be more intuitive if UI elements with the same purpose or function were pre-grouped and users could select them all at once. Considering feedback, we anticipate the UI selection interface can improve by allowing app developers to provide selectable UI groups via simple APIs in the near future.

Below, we share a number of remarkable quotes from the participants for a better understanding of the user experiences of each scenario.

*Scenario 1: LiveMe. "when using multiple devices, the broadcast screen was not overlapped by the chat keyboard, which made it convenient to watch the broadcast and chat live at the same time (P1)"*. Similarly, *"Applying this technology to smart TVs could be incredibly beneficial because they typically lack a typing interface (P5)"*.

*Scenario 2: PaperDraw. "I think moving the painting tools and colors to another screen seemed to be a necessary feature for video editors and designers. Additionally, using the color UIs on the remote device provided a similar experience to using a palette when painting (P6)*". On the other hand, P7 commented *"The use case seemed useful, but in actual use, I had to choose target UIs one by one to distribute them, which makes it feel cumbersome in terms of UX design"*.

*Scenario 3: Naver map. "While many navigation apps offer the ability to search for a destination route using voice recognition, it is still very inconvenient due to its low accuracy and slow responsiveness. On the other hand, I think FLUID is a very competitive technology because it enables a passenger to search for the destination on behalf of the driver by moving the navigation app's search box to his/her device quickly (P10)"*.

*Scenario 4: VLC Player. "Typically, we can control a smart TV with a remote controller, but it is still inconvenient to interact with several UIs on the smart TV using the remote controller. So I think the UI distribution technology is very suitable for the smart TV environment. If I can move a seek bar on the smart TV to my smartphone, it will be more intuitive to control the video content like this scenario (P11)"*.

In summary, the results of this user study demonstrated that the multi-device interactions empowered by FLUID can be well accepted by users in terms of performance, intuitiveness, usability, and satisfaction. In particular, we observed that the participants successfully accomplished various task scenarios in the multi-device environment by intuitively selecting necessary UIs through the UI selection interface and transferring them to other devices.

## XIII. RELATED WORK

**App-level multi-surface support.** To support multi-surface use, several custom applications have been released, such as Netflix, Google Docs, and so on. These types of apps support data sharing based on cloud services, and thus users can access the same content from multiple devices. This allows users to continuously conduct tasks across different devices or simultaneously collaborate with other users. However, this method incurs considerable costs to create custom apps and has low applicability because such custom apps are developed to support only specific multi-surface scenarios. However, FLUID adopts a system-level approach that can support various legacy apps without modifying their source codes.

**New programming models/tools.** Several studies have suggested new programming models or development tools for developing multi-surface apps. UIWear [2] presented a development framework that automatically creates a new smartwatch app with UIs extracted from a smartphone app.

CollaDroid [3] proposed a development tool that inserts source code for UI distribution into a legacy app allowing a user to conduct collaborative tasks with others. However, these methods do not consider extracting app-specific graphical states related to custom UIs, and thus they cannot provide the same look-and-feel for custom UIs of legacy apps. By contrast, FLUID can identify the graphical states of custom UIs with no false-negative errors by analyzing the UI source code of both the Android platform and target app. This allows FLUID to provide greater applicability to various legacy apps.

**Screen sharing.** Many applications support screen sharing (or screen mirroring), which copies the screen of one device onto another device, e.g., Teamviewer [27], Scrcpy [11], AirPlay [4], and Chromecast [5]. However, as this method replicates screens in a coarse-grained manner (i.e., at a unit of the entire screen), its use cases are limited in utilizing the benefits of multiple surfaces from the user's perspective. For example, screen sharing cannot support the use cases of FLUID, in which an app distributes different UI elements across multiple surfaces. As an exception, AirPlay and Chromecast can deliver partial UIs to other devices only for some specialized apps; therefore, they cannot support general legacy apps unless the apps are developed for Chromecast or AirPlay.

**App/thread migration.** Flux [6] is an app migration mechanism that allows an Android app to move to another device on runtime to utilize its surface. However, Flux allows an app to utilize only one surface at a time, making it impossible to utilize multiple surfaces concurrently by distributing different UIs across them, in contrast to FLUID. Many studies have proposed thread migration techniques [28], [29] to support the offloading of computation-intensive tasks for servers with powerful computing resources to improve their performance. However, these techniques do not focus on design issues for offloading UI tasks to support interactive usage, which is one of the contributions of FLUID.

**Cross-device RPC.** RPC is a well-known technique that has been used in various distributed systems for decades and contains several API libraries, like Java RMI [30], etc. However, unlike FLUID, it is not applicable to unmodified legacy apps because it requires apps to use specific RPC libraries. Mobile Plus [31] is the closest approach to the proposed FLUID as it allows legacy apps to use RPC by transparently extending within-device function calls to cross-devices. However, Mobile Plus extends *inter-process* method calls (i.e., binder calls) to a cross-device RPC, whereas FLUID extends *intra-process* method calls within the same process boundary. Such major differences lead to new challenges not considered by Mobile Plus, such as intercepting local method calls and ensuring the accuracy of replicated RPCs.

**Cross-device I/O Sharing.** Several studies have been conducted to support I/O sharing between multiple devices. Rio [32] provides virtualization at the device file layer to enable I/O resource sharing. M2 [33] proposed a data-centric approach that uses high-level device data to enable I/O sharing between heterogeneous devices. MobiUS [34] supports screen sharing between devices for high-resolution video. However, these studies do not consider the use of multiple surfaces in a fine-grained manner (i.e., at the unit of UI elements), which is the key difference compared to FLUID.

## XIV. Discussion

**Direct memory writes to the UI objects.** The current design of FLUID cannot synchronize UI changes caused by direct memory writes to the graphical states of UI objects, that is, direct memory writes to their public fields. However, these cases are rarely observed in legacy apps because most app developers avoid using such memory writes, which may lead to race conditions during the UI rendering process. In general, they follow the conventional principle of encapsulating the graphical states of UI objects and changing them using only function calls [35]. We can enable memory writes in public fields of UI objects using the field-level distributed shared memory technique proposed in COMET [29].

**Native object serialization.** Because C++ does not provide runtime type information, such as Java Reflection, it is impossible to serialize native objects implemented in C++ unless app developers utilize external serialization libraries. Therefore, the current prototype of FLUID includes serialization logic only for the native objects of Android C++ graphics libraries (e.g., Skia [36]), but not for third-party native libraries. We observed that the 20 legacy apps (Table I) use custom Java objects for UIs but no custom native objects; thus, the current working prototype successfully supports them. Moreover, we examined 43 open-source apps with an average of 4,505 star scores on GitHub and found that only one of the apps used a custom C++ library for graphics. This means that most apps tend to employ the standard C++ graphics libraries of Android.

**Multi-surface layouts.** During the coverage tests described in Section XI-A, we found some cases where their usability could be improved with some assistance from app developers. Specifically, with the current FLUID prototype, a user can select only visible UI elements for migration or replication, which may sometimes lead to a situation far from the user's expectation. For example, an app called *Maps.me* makes a relevant query window visible only after a user types a query in a text input box, due to which we cannot migrate or replicate the two UIs together for better usability before the user starts typing. However, this limitation can be addressed if app developers bind the relevant query window and text input box into one container UI (i.e., a layout object). In addition, when rendering guest UIs, FLUID enables the renderer thread on a guest device to optimize their size to fit the guest surface resolution for better usability. Because this method does not modify the graphical states of the guest UIs, we can guarantee the deterministic execution of FLUID, as described in Section VII-B. However, if app developers offer layout guidelines for guest surfaces, they can optionally control how FLUID will display guest UIs on other devices. For example, if a user migrates a button UI that is set to be placed at a position relative to other UIs on a host device, the current FLUID prototype disregards the configuration and places the button at an arbitrary position on the guest screen (e.g., the middle of the screen). The layout guidelines can better address such issues that may harm app usability.

**Unsupported legacy apps.** FLUID cannot support some legacy apps, such as 3D games or web applications, that utilize third-party UI frameworks (e.g., Unity [37] or WebKit [38]) because their UI objects are managed by internal data structures of these frameworks. This feature makes it impossible for FLUID to access UI objects and transparently transform their

function calls into RPCs. However, third-party frameworks handle UI objects internally in a similar manner as Android, i.e., their UI objects are also managed by tree structures and updated by local functions. Therefore, if we can modify them, the general design of FLUID can be comprehensively applied.

**Support for dynamically changing UIs.** FLUID may encounter frequent updates when migrating dynamically changing UIs in real-time applications (e.g., video streaming), which leads to intolerable network latency for users. To alleviate this overhead, FLUID employs a more efficient approach by transmitting only data (e.g., RPC arguments) for a few updated UIs, as opposed to sending full-screen images like mirroring. This can result in a substantial reduction in network transmission. Furthermore, we anticipate that FLUID will support such dynamic UIs with better performance by capitalizing on upcoming wireless standards featuring high bandwidth and ultra-low latency. For example, 6G networking technology [39] is expected to provide 1-microsecond latency and 1-Tbps bandwidth, which facilitates handling frequent updates for dynamic UIs. Also, FLUID can be extended to implement video encoding/compression to reduce such update overhead.

**Security issues in multi-surface computing.** In this work, we assume that both host and guest sides are reliable; however, either may be compromised by real-world attackers. For instance, malicious apps on the guest might attempt to intercept user-private data displayed on migrated UIs. To address this, we can run the FLUID wrapper app in a trusted execution environment like ARM TrustZone [40], ensuring secure execution and rendering of migrated UIs. Conversely, we can also think of a situation where a malicious host app accesses guest-side data without authorization. To avoid this, FLUID restricts the accessible range of migrated UIs to the wrapper app's internal storage, which stores only the necessary code and resource files for UI rendering. This prevents the malicious host app from leaking or tampering with user-private data stored elsewhere on the guest side.

**Multi-device interaction beyond WLAN.** To interact with remote devices in different WLANs, FLUID can use a dedicated proxy server, as other network services do. However, using a proxy server inevitably leads to increased network delay, which may degrade the overall performance of FLUID. So, we need to apply several techniques such as compression or network data caching to reduce the performance overhead.

**Applying FLUID to other systems.** Thanks to the general design principles of FLUID, it can be applicable to Android, as well as other mobile platforms. FLUID is designed based on the assumption that a single UI thread manages UIs in a hierarchical structure and updates their states through function calls. This is a typical design paradigm for most GUI-based systems (e.g., UIView [41] in iOS). To apply FLUID design to other mobile platforms, we have to consider the following technical issues. *i) Finding the graphical states of each UI object through static analysis.* Unlike Android, other mobile platforms may support apps compiled directly into binary code. In this case, it is fundamentally difficult to analyze a binary code using CHA. However, if app developers utilize the CHA feature offered by compilers (e.g., Swift SIL optimizer [42]), FLUID can easily employ its result. *ii) Serializing the graphical states of the UI objects.* It is infeasible to support UI object serialization if other mobile platforms do not provide runtime type information. However, as explained earlier, this problem can be solved if compilers are extended to provide such information. *iii) Supporting transparent cross-device RPC.* If other mobile platforms are not VM-based, unlike Android, it is difficult to intercept function calls directly. Instead, we can use code instrumentation techniques that insert extra code into an app to create and transfer RPC messages.

**Compatibility between heterogeneous devices.** FLUID can support successful multi-surface interaction across devices with varying form factors (e.g., smartphone or tablet) as long as they share the same Android version and vendor. However, compatibility issues may arise when their platform versions or vendors differ, as some UI classes can be defined differently on each device. To mitigate this, we can leverage a translation layer to make migrated UIs compatible with guest devices, as discussed in Section XI-F. This approach is possible because the differences in UI classes are not large enough to compromise the applicability of FLUID, and their underlying UI mechanism is the same. On the other hand, it is worth noting that FLUID has a limitation in its applicability for heterogeneous platforms—it cannot support interaction between Android and non-Android platforms (e.g., iOS) because each utilizes a completely different execution environment for UI rendering. To support such an interaction, we can consider another approach that converts distributed mobile UIs into highly portable web UIs consisting of HTML and Javascript. We leave it as future work.

## XV. Conclusion

This paper introduces FLUID, a new mobile platform based on Android, which supports innovative methods of interacting with unmodified legacy applications across multiple devices. FLUID allows users to migrate or replicate UI elements to multiple devices selectively while facilitating cross-device cooperation between the app logic and distributed UI objects transparently and deterministically. Our experiments with the FLUID prototype demonstrated that it achieves high flexibility, transparency, and applicability in multi-surface usage, attracting a broad range of legacy apps in multi-surface environments. In the near future, we anticipate that FLUID will facilitate the development of useful and creative multi-surface apps while providing various multi-device user experiences.

## References

[1] T. Spangler, "U.S. Households Have an Average of 11 Connected Devices - And 5G Should Push That Even Higher," 2019, https://variety.com/2019/digital/news/u-s-households-have-an-average-of-11-connected-devices-and-5g-should-push-that-even-higher-1203431225/.

[2] J. Xu, Q. Cao, A. Prakash, A. Balasubramanian, and D. E. Porter, "Uiwear: Easily adapting user interfaces for wearable devices," in *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '17, 2017.

[3] J. Zheng, X. Peng, J. Yang, H. Cai, G. Huang, Y. Zhang, and W. Zhao, "Colladroid: Automatic augmentation of android application with lightweight interactive collaboration," in *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, ser. CSCW '17, 2017.

[4] Apple, "Airplay," 2023, https://store.google.com/product/chromecast.

[5] Google, "Chromecast," 2019, https://www.apple.com/airplay/.

[6] A. Van't Hof, H. Jamjoom, J. Nieh, and D. Williams, "Flux: Multi-surface computing in android," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15, 2015.

[7] S. K. Card, A. Newell, and T. P. Moran, *The Psychology of Human-Computer Interaction*. L. Erlbaum Associates Inc., 1983.

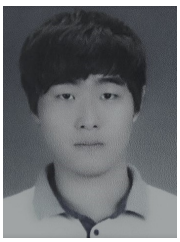
[8] J. Nielsen, *Usability Engineering*. Morgan Kaufmann, 1994.
[9] S. B. Shneiderman and C. Plaisant, *Designing the user interface*. Pearson Addison Wesley, 2005.
[10] ClockworkMod, "Vysor," 2019, https://www.vysor.io/.
[11] Genymobile, "Scrcpy," 2019, https://github.com/Genymobile/scrcpy.
[12] Expedia, "Expedia," 2019, https://www.expedia.com/app.
[13] A. Sahami Shirazi, N. Henze, T. Dingler, K. Kunze, and A. Schmidt, "Upright or sideways? analysis of smartphone postures in the wild," in *Proceedings of the 15th International Conference on Human-Computer Interaction with Mobile Devices and Services*, ser. MobileHCI '13, 2013.
[14] S. R. Group, "Soot - a java optimization framework," 2019, https://github.com/Sable/soot.
[15] E. Software, "Kryo," 2019, https://github.com/EsotericSoftware/kryo.
[16] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao, "Accelerating mobile applications through flip-flop replication," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '15, 2015.
[17] Oracle, "Jdk swing framework," 2018, http://docs.oracle.com/javase/6/docs/technotes/guides/swing/.
[18] T. Q. Company, "The qt framework," 2019, https://www.qt.io/.
[19] Apple, "Macos cocoa," 2019, http://developer.apple.com/technologies/mac/cocoa.html.
[20] S. Zhang, H. Lü, and M. D. Ernst, "Finding errors in multithreaded gui applications," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 243–253.
[21] P.-E. Danielsson, "Euclidean distance mapping," *Computer Graphics and Image Processing*, 1980.
[22] Y. Z. Liwei Wang and J. Feng, "On the euclidean distance of images," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2005.
[23] D. Katz and D. Ward, "Bidirectional Forwarding Detection (BFD)," RFC 5880, Jun. 2010. [Online]. Available: https://www.rfc-editor.org/info/rfc5880
[24] Google, "Measure app performance with android profiler," 2019, https://developer.android.com/studio/profile/android-profiler.
[25] M. Solutions, "Monsoon power monitor," 2019, https://www.msoon.com/.
[26] Apple, "ipad pro - technical specifications," 2021, https://www.apple.com/ipad-pro/specs/.
[27] TeamViewer, "Teamviewer - remote support, remote access, service desk, online collaboration and meetings," 2018, https://www.teamviewer.com.
[28] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11, 2011.
[29] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code offload by migrating execution transparently," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.
[30] E. Pitt and K. McNiff, *Java.Rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., 2001.
[31] S. Oh, H. Yoo, D. R. Jeong, D. H. Bui, and I. Shin, "Mobile plus: Multi-device mobile platform for cross-device functionality sharing," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '17, 2017.
[32] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong, "Rio: A system solution for sharing i/o between mobile systems," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14, 2014.
[33] N. AlDuaij, A. Van't Hof, and J. Nieh, "Heterogeneous multi-mobile computing," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '19, 2019.
[34] G. Shen, Y. Li, and Y. Zhang, "Mobius: Enable together-viewing video experience across two mobile devices," in *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, ser. MobiSys '07, 2007.
[35] Z. Mednieks, L. Dornin, G. B. Meike, and M. Nakamura, *Programming Android*. Oreilly & Associates Inc, 2011.
[36] Skia, "Skia graphics library," 2019, https://skia.org/.
[37] "Unity Real-Time Development Platform," 2021, https://unity.com/.
[38] "WebKit: A fast, open source web browser engine," 2021, https://webkit.org/.
[39] M. Giordani, M. Polese, M. Mezzavilla, S. Rangan, and M. Zorzi, "Toward 6g networks: Use cases and technologies," *IEEE Communications Magazine*, 2020.
[40] Arm, "Trustzone - arm developer," 2021, https://developer.arm.com/ip-products/security-ip/trustzone.
[41] N. Smyth, *iOS 12 App Development Essentials*. Payload Media, Inc., 2012.
[42] Apple, "Welcome to swift.org," 2019, https://swift.org/.


**Sangeun Oh** is an Assistant Professor with the Department of Software and Computer Engineering, Ajou University, Suwon, South Korea. His research interests include mobile/IoT systems and real-time embedded systems. He received the B.S. degree in computer and communication engineering from Korea University, Seoul, South Korea in 2012, and the M.S. and Ph.D. degrees in computer science from KAIST, Daejeon, South Korea, in 2014 and 2020, respectively.

**Ahyeon Kim** received the B.S. degree in electrical engineering and the M.S. degree in computer science from KAIST, Daejeon, South Korea, in 2019 and 2021, respectively.

**Sunjae Lee** received the B.S. and M.S degrees in computer science from KAIST, Daejeon, South Korea, in 2019 and 2021, respectively. He is currently pursuing the Ph.D. degree in computer science at KAIST, Daejeon, South Korea.

**Kilho Lee** Kilho Lee is an assistant professor with the School of AI Convergence, Soongsil University, South Korea. He received his BSc degree in Information and Computer Engineering from Ajou University, and his MSc and PhD degrees in Computer Science from KAIST. His interests include system design for real-time embedded systems and cyber-physical systems.



**Dae R. Jeong** Dae R. Jeong received the B.S., and M.S. degree from the School of Computing, KAIST, Daejeon, South Korea, in 2014, 2016 respectively. He is a Ph.D candidate in the School of Computing, KAIST. His current research interests include finding and diagnosing concurrency bugs under various memory models in a large system software including a kernel.

**Steven Y. Ko** is an Associate Professor and a Dean's Excellence Fellow in the School of Computing Science at Simon Fraser University. His research interest is improving the reliability and security of mobile systems. He received a B.S. degree in Mathematics from Yonsei University, an MS in Computer Science and Engineering from Seoul National University, and a PhD in Computer Science from University of Illinois at Urbana-Champaign.

**Insik Shin** is a professor in the School of Computing at KAIST, Korea. His research interests include real-time embedded systems, systems security, mobile computing, and cyber-physical systems. He received a Ph.D. degree from the University of Pennsylvania, USA, an MS degree from Stanford University, USA, and a BS degree from Korea University, Korea, all in Computer (& Information) Science.