The Journal of Systems and Software 119 (2016) 31–44

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Software architectural principles in contemporary mobile software: from conception to practice

Hamid Bagheri[a,*], Joshua Garcia[a], Alireza Sadeghi[a], Sam Malek[a], Nenad Medvidovic[b]

[a] School of Information and Computer Sciences, University of California, Irvine, United States
[b] Computer Science Department, University of Southern California, United States

ARTICLE INFO

Article history:
Received 3 May 2015
Revised 23 May 2016
Accepted 25 May 2016
Available online 1 June 2016

Keywords:
Software architecture
Android
Architectural styles

ABSTRACT

The meteoric rise of mobile software that we have witnessed in the past decade parallels a paradigm shift in its design, construction, and deployment. In particular, we argue that today's mobile software, with its rich ecosystem of apps, would have not been possible without the pioneering advances in software architecture research in the decade that preceded it. We describe the drivers that elevated software architecture to the centerpiece of contemporary mobile software. We distill the architectural principles found in Android, the predominant mobile platform with the largest market share, and trace those principles to their conception at the turn of century in software architecture literature. Finally, to better understand the extent to which Android's ecosystem of apps employs architectural concepts, we mine the reverse-engineered architecture of hundreds of Android apps in several app markets and report on those results.

1. Introduction

Mobile computing has come a long way from a decade ago. Development of mobile software used to be an art exercised by a few, savvy, experienced developers, capable of hacking low-level C code—the *lingua franca* of mobile software at the time. The resulting software systems were often monolithic, rigid, one-off programs, which were hard to construct, understand, and maintain (Picco et al., 2014). Although software architectural principles had found widespread use in structuring the traditional desktop software at the turn of century (Taylor et al., 2009), mobile software was often devoid of such structures (Picco et al., 2014; Medvidovic et al., 2003).

The dominant preconception was that for developing efficient software, suitable for deployment on resource-constrained mobile platforms, it is necessary to compromise on flexibility and decoupling achieved through architectural principles, such as decomposition of a software system into components, separation of communication links in the form of connectors, and so on Medvidovic et al. (2003); Malek et al. (2007). In particular, programming-language abstractions needed for the realization of those architectural concepts were deemed unsuitable for use in mobile software.

Today's mobile software, however, differs greatly from that of a decade ago (Wasserman, 2010). Our empirical investigation—the details of which are described in Section 5—shows that software architecture plays a significant role in the development of modern mobile software. Many of the ideas devised in pioneering software architecture work, developed around the turn of this century, have found a home in the contemporary mobile software. In particular, Android, which is the predominant mobile platform, realizes many of the architectural principles previously advocated by the software-engineering community.

At first blush, one may conjecture that the increasing prominence of software architectural principles is a natural progression of software-engineering practices in any computing domain. But when we look at other closely related areas of computing, such as embedded software, we do not find a similar adoption of software architectures. It is, thus, important to understand the drivers behind the rapid adoption of software architectures in mobile computing, as well as the nature of the adopted architectural concepts and principles, and how their use has impacted the development of mobile software.

To that end, we first describe several requirements that drove the adoption of many of the architectural principles advocated in the literature in modern mobile software development. We also trace back those principles to their conception in the pioneering software-architecture research, in particular the research on the

* Corresponding author.
E-mail addresses: hamidb@uci.edu (H. Bagheri), joshug4@uci.edu (J. Garcia), alirezs1@uci.edu (A. Sadeghi), malek@uci.edu (S. Malek), neno@usc.edu (N. Medvidovic).

http://dx.doi.org/10.1016/j.jss.2016.05.039
0164-1212/

applicability and benefits of architecture-based design and development in a mobile setting. Afterwards, we present some of the key architectural concepts found in Android, often codified in its *application development framework*, which provides programming-language constructs for architecture-based development of mobile software, including support for the realization of software components, connectors, events, configurations, and architectural styles.

We argue that software architectural support in Android has played a key role in its meteoric rise and success. Our empirical observation corroborates the notion that architectural building blocks in Android are supported in the Android platform's programming constructs, which promises to dramatically improve an app developer's productivity, and also makes it much easier to develop complex apps without a formal education in programming, or previous programming experience. This, in turn, supports even novice programmers in developing sophisticated apps with the potential of becoming popular in app markets, such as Google play. Codification of an architectural family and separation of communication from the application logic, using asynchronous connectors, have facilitated the integration of third-party code in a mobile device, thereby directly spawning a vibrant ecosystem of apps.

The contributions of this paper can be summarized as follows:

- Identifies the drivers behind the rapid adoption of software architecture concepts and principles in contemporary mobile software, specifically Android.
- Distills the architectural principles found in Android and illustrates them using a popular mobile app.
- Traces back those principles to their conception in software-architecture research.
- Reports on the characteristics of architectures found in the Android ecosystem of apps by mining hundreds of Android apps in several app markets.
- Reflects on deviations from how architectural concepts have been prescribed in architecture literature and the manner in which Android has realized some of those concepts, thereby concluding with lessons that could be of interest to both the mobile-computing industry as well as software-architecture researchers.

The remainder of the paper is organized as follows. Section 2 outlines the mobile-computing requirements that drove the adoption of software architectures. Section 3 describes a popular mobile app that we use to illustrate the architectural concepts in Android. Section 4 presents the key architectural principles followed by Android as well as their conception in the literature that predates it. Section 5 reports on the architectural properties of hundreds of reverse-engineered apps. Section 6 discusses the salient outcomes of our study. Finally, the paper concludes with an outline of the related research in Section 7 and an overview of our contributions in Section 8.

2. Mobile computing drivers

Before describing the architectural concepts found in contemporary mobile software, it is important to understand the key challenges that the mobile-computing industry has had to overcome over the past decade. The need to overcome these challenges is the root cause of the drastic shift toward the adoption of software architectures in today's mobile software.

(D1) App ecosystem. Perhaps the most striking difference between today's mobile platforms and those of a decade ago is the notion of *app ecosystem*. An app ecosystem is the interaction of a set of independently developed software elements (apps) on top of a common computing platform that results in a number of software solutions or services (Manikas and Hansen, 2013; Bosch, 2009). App stores and apps have changed the landscape of mobile computing: entrepreneurs are able to reach a large consumer market, consumers can choose from thousands of apps at a nominal cost, and app advertising has created a lucrative form of revenue for the developers. Apps extend the capabilities available on a platform, making the platform more attractive to the users. Therefore, a vibrant app ecosystem is crucial to the success of a mobile platform, such as Android. A key challenge in conceiving such platforms, however, was encoding constraints and rules to enable a properly functioning ecosystem with certain norms of structure and behavior, yet remaining sufficiently flexible to allow the developers to fully exploit the capabilities available on modern mobile devices (Eklund and Bosch, 2014).

(D2) Developer productivity. As alluded to earlier, the development of mobile software previously involved low-level programming, often against the various device drivers, akin to the kind of practices still followed in the embedded-computing domain (Malek et al., 2007). At the same time, the success of an app ecosystem, and thus the corresponding mobile platform, hinges on the availability of a large number of apps for end users to choose from. Such an app ecosystem requires a large pool of qualified developers capable of creating apps without highly specialized skills. Thus, the awareness grew that mobile platforms vying for a vibrant app ecosystem need to provide the developers with high-level implementation abstractions, and properly-enforced rules and constraints on how those abstractions can be composed, to ease the construction of apps.

(D3) Interoperability. A particular challenge in conceiving the modern mobile-computing platforms lied in providing a rich user experience, where a mobile device's native capabilities (e.g., phone, camera, and GPS) as well as third-party apps are able to integrate and interact with one another. Achieving this objective requires interoperability between third-party apps that are developed independently, and possibly without knowledge of one another, as well as software and hardware services that are available on a multitude of proprietary devices (Ebert and Jones, 2009). This challenge called for explicit specification of exposed interfaces of apps, as well as standards, rules, and architectural styles that regulate the interactions of apps and system services.

(D4) Security and privacy. Seamless interoperability between apps, together with the various private user data collected on modern mobile devices, gave prominence to security and privacy issues (La Polla et al., 2013). In addition, the app-store model of provisioning apps proved convenient not only for the end users, but also for the malware writers that exploited it for delivering malicious code into the users' devices (Zhang et al., 2013). To combat these threats, proper abstractions were needed for specification, assessment, and enforcement of security properties (e.g., information flow and access control) at a higher level of granularity than code.

(D5) Resource constraints. Finally, as the apps deployed on mobile platforms continued to grow in size and complexity, resource constraints (e.g., energy and memory) continued to pose an ever-present challenge. Specifically, there was a need to manage and coordinate the resources consumed by third-party apps (Nikzad et al., 2014). As an example, consider that many apps may require access to GPS information, but an uncoordinated access to such information rapidly drains the battery of a mobile device. Similarly, multiple apps may be running on a mobile device, but at any point in time only parts of those apps are actively used; without dynamically offloading the unused elements at runtime, a device would run out of resources rapidly.[1] To address these chal-

[1] Note that in this paper the term "offload" is used to mean temporarily removing a task from processor/memory to make them available for other tasks. It should not

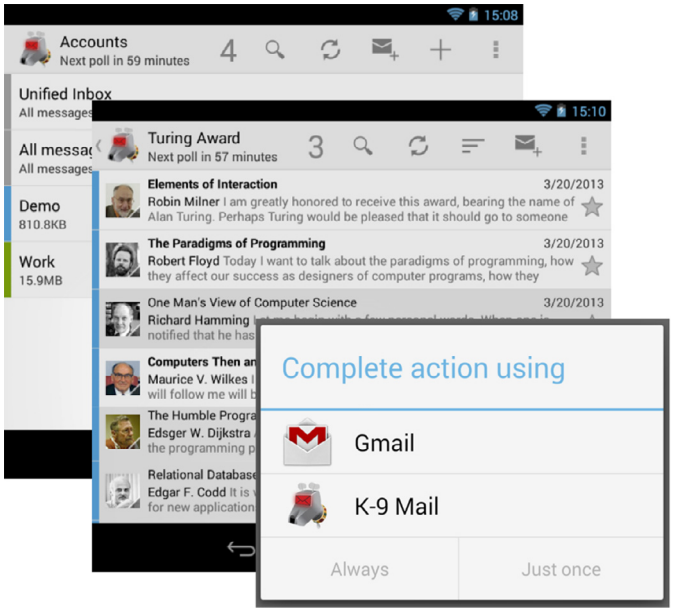

Fig. 1. K-9 mail client app.

lenges, there was a need for structural and behavioral abstractions at a higher-level of granularity than code that could effectively mitigate the non-functional concerns in a coordinated fashion.

3. Running example

To illustrate the concepts in this paper, we use a popular email client for Android, namely K-9 mail. Fig. 1 provides a screen shot of K-9 mail, which is an app that provides the ability to send and receive messages from multiple email accounts and integrate with Firefox. For example, K-9 mail can send an email after a user clicks on a hypertext email link embedded in a web page (`<a href=''mailto:...'' >`). Moreover, K-9 mail uses OpenPGP, the most widely used email encryption standard, for encrypting and decrypting emails.

Fig. 2 shows the architecture of K-9 mail that we have reversed engineered from its source code. The details of this architecture will be explained in the following sections. Note that the complete architecture of K-9 mail is significantly more complex and comprises more than 40 software components. For clarity, we only show a subset of its most interesting elements. K-9 mail also interfaces with two other apps (Firefox Browser and OpenPGP), the detailed architectures of which have been elided due to space limitations.

4. Architectural principles in Android

We now describe the software architectural concepts and principles that have played a significant role in addressing the challenges described in Section 2, and illustrate them using the architecture of K-9 mail.

4.1. Software architecture building blocks

In any software project, implementation of the software is preceded with a design activity. The design process could be either formal, whereby the design decisions are documented in some preliminary architectural models, or informal, whereby the designer simply draws an initial decomposition of the system on a whiteboard. Regardless of the approach followed, the key to this process is how the developer conceptualizes a software system, i.e., the design idioms and elements a developer employs to decompose a system into its constituent parts.

Object-oriented constructs, such as classes, are typically too low-level for conceptualizing the high-level structure of a complex software system. The pioneering work of Wolf and Perry (1992), as well as Shaw and Garlan (1996), showed that a proper abstraction for reasoning about the elements of a software system are its *Components*, *Connectors*, and their *Configurations*. Subsequent research on Prism (Medvidovic et al., 2003; Malek et al., 2005; 2007), an architectural design framework for mobile software, showed that these abstractions also provide a proper level of granularity for the design and construction of mobile software.

In practice, many of the same constructs devised by the software-architecture research community can be observed in the Android framework. *Components* are also the basic building blocks of Android apps Google. Android provides four types of components:

1. **Activity** provides the basis of the Android user interface Google. Each app may have multiple Activities representing different screens of the app to the user. The *Accounts* component in Fig. 2 is an example of an Activity that allows the user to view the status of its mailbox accounts, and corresponds to the bottom-most screen depicted in Fig. 1.
2. **Service** provides the background processing capabilities, and does not provide any user interface Google. The *MailService* component in Fig. 2 is an example of a service that periodically downloads mail from the server, which can occur while a user interacts with another app.
3. **Content provider** offers data storage and retrieval capabilities to other components Google. It can be used to share and persist data within components of an app, as well as across apps. *EmailProvider* in Fig. 2 is an example of a Content Provider; it maintains a repository of downloaded emails on a device.
4. **Broadcast receiver** responds asynchronously to system-wide message broadcasts Google. A Broadcast Receiver typically acts as a gateway to other components, and passes messages to Activities or Services for handling. *RemoteControlReceiver* is an example of a Broadcast Receiver in Fig. 2, and it dispatches Intents for externally controlling the behavior of K-9. Although classified as a component type in Android, we argue that Broadcast Receiver is best explained as a connector; this is an issue that we revisit later in this paper.

Android also supports lower-level components, called *Fragments*, that provide another layer of abstraction within an Activity Google. A Fragment serves as a modular section of an Activity, responsible for handling events associated with a particular part of the corresponding user interface. Each Fragment has its own lifecycle, receives its own events, and can be added or removed while its enclosing Activity is running Google.

Components and whole apps in Android communicate with one another using a variety of mechanisms. Although the Android literature does not refer to them as connectors, Android's runtime environment provides a variety of connector types. Our study revealed four types of connectors (Mehta et al., 2000) in Android:

1. **Explicit message-based** connectors enable the exchange of *explicit Intents* using Android's *inter-process communication (IPC)* mechanism. An Intent message consists of a payload as well as meta-data describing it Google. The meta-data includes *action, data*, and *category*, specifying the general action to be performed by the recipient component, the type of payload, and

be confused with the usage of the term to mean moving computations to a different platform.

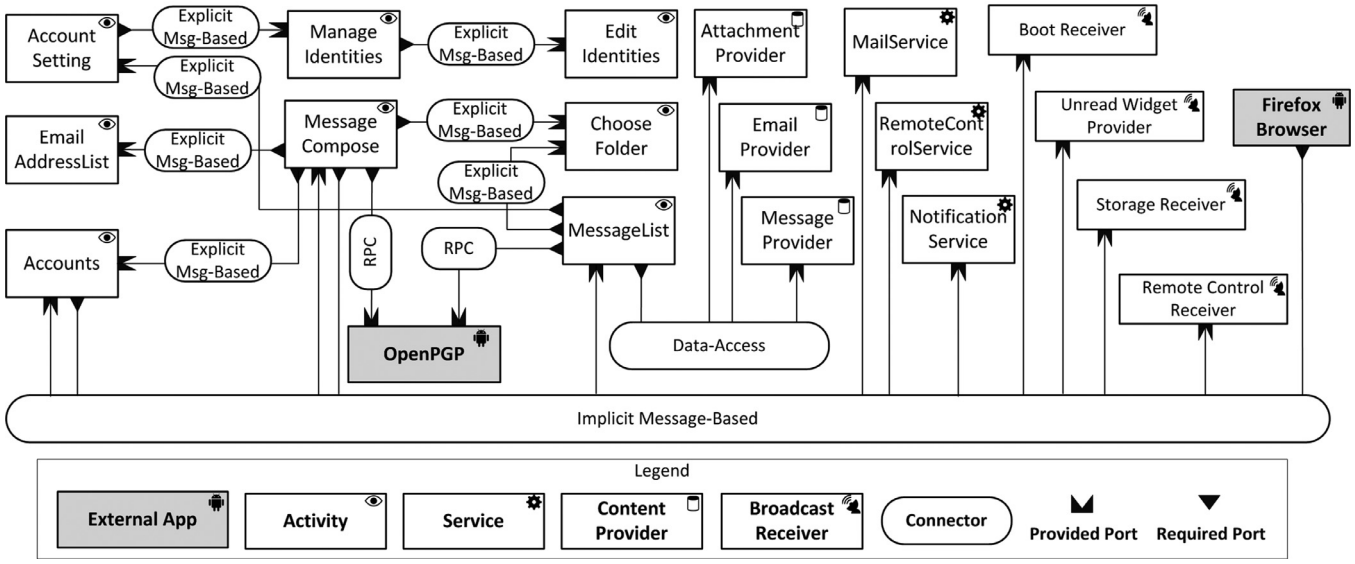

Fig. 2. K-9 mail Android app architecture.

the kind of component that should handle the Intent, respectively. An explicit Intent is one where the message is tagged with its recipient Google.

2. **Implicit message-based** connectors enable the exchange of *implicit Intents* using Android's IPC mechanism. An implicit Intent is one where the message is not tagged with its intended recipient Google. Rather, its meta-data (i.e., action, data, category) is matched against the *Intent Filters* specified by other components. Intent Filters are Android's way of specifying the provided interfaces of a component. An Intent Filter describes the kinds of requests a given component can respond to. Interestingly, however, Android does not provide a mechanism to specify required interfaces; we will revisit the implications of this later in the paper.
3. **Data access** connectors correspond to Android's `Content Resolver` Google, which provides an interface for adding, removing, and querying the stored data in a Content Provider.
4. **Remote procedure calls (RPC)** provide IPC through method-invocation interaction using *stubs*, which are automatically generated from the specification of a component's interfaces in Android's Interface Definition Language Google.

Finally, Android provides a separate XML file, called *manifest*, that accompanies each app Google. This file allows for specification of the system's configuration in terms of its components and interfaces. We will provide a more detailed discussion of manifest file in Section 4.4.

By providing a common vocabulary of architectural constructs that developers can use in designing their apps, Android has provided a level of uniformity in the structure and behavior of apps, while leaving flexibility in how those constructs are composed **(D1)**. Moreover, the separation of computation, in the form of developer provided components, from communication, in the form of reusable platform-provided connectors, has alleviated the interoperability issues **(D3)**.

4.2. Hierarchical (de)composition

The work of Kruchten (1995) underlined the importance of views and viewpoints in mitigating the complexity of architectures. Hierarchical decomposition of a software system's architecture, and hierarchical composition of a system's components, are fundamental tools for producing views of a system's architecture at different levels of abstraction (Taylor et al., 2009). The components of a given conceptual unit are grouped together into a larger, more complex component; subsequently, that component may be grouped with other like components into even larger components. Depending on the task at hand, stakeholders are able to focus on the system's architecture at one level of abstraction or another. The key to achieving hierarchies in architecture is the ability to mask the interfaces of the lower-level components through granular higher-level components. Generally, composition has been achieved in two different ways in the architecture literature: behavioral and structural (Taylor et al., 2009). As an example of the former, consider service-oriented architectures, where services are orchestrated using a work-flow language (e.g., BPEL) to realize a higher-level service, also known as a *composite service* (Erl, 2006). As an example of the latter, consider Prism (Malek et al., 2005), where a component at one layer of abstraction may itself consist of an internal architecture with lower-level components.

Android's approach to hierarchical (de)composition is structural in nature, and achieved in three key ways: (1) a top-level architecture containing a set of apps, (2) an app architecture containing its own components and connectors, and (3) an Activity containing a set of Fragments. This hierarchical decomposition enables independent development of modules at different levels of abstractions (**D1**). It further promises to improve developer's productivity (**D2**), because by tackling complexity through hierarchical decomposition, a developer can focus on the appropriate abstractions for the current tasks she is undertaking.

If we consider the app software running on a mobile device as our top-level architecture, each app is essentially a top-level component. Apps at this level can communicate with one another using Intent messaging connectors or RPCs. Thus, the device (e.g., a phone) at any given point in time has a running architecture. This architecture is dynamic, as new apps are installed, executed, stopped, and removed.

However, each app itself is also composed of an architecture, meaning that each app comprises components and connectors described in the previous section. An app may itself make use of other apps. Components comprising an app may also extend one another to support specialization of components. Fig. 2 depicts an architecture at the level of an app, in this case that of K-9 mail.

As mentioned in the previous section, at the lowest level of granularity, Android provides *Fragments*. Activities

Table 1
Architectural styles of Android apps

Styles	Components	Connectors	Sync	Benefits	Drivers
Message-based implicit invocation	Activities, Services	Implicit msg-based	Async	Evolvability, Maintainability, Heterogeneity	D2, D3, D5
Publish-subscribe	Activities, Services, Broadcast Receivers	Implicit msg-based	Async	Evolvability, Maintainability, Heterogeneity, Eefficiency	D2, D3, D5
Message-based explicit invocation	Activities, Services	Explicit msg-based	Async	Evolvability, Maintainability, Heterogeneity, Effficiency	D2, D3, D5
Shared state	Content providers	Data access	Both	Efficiency, Maintainability	D2, D5
Distributed objects	Activities, Services, External Apps	RPC	Both	Interoperability, Maintainability	D2, D3

encapsulate Fragments through well-defined interfaces exposed at the Activity level Google. As an example, K-9's `MessageList` component from Fig. 2 contains two Fragments: `MessageListFragment`, which depicts the current list of messages, and `MessageViewFragment`, which depicts the current message being viewed.

Fragments entail reusability (**D2**) at a lower level of abstraction than screens, but at a higher level than UI widgets. For example, on a larger screen (e.g., on a tablet), `MessageListFragment` and `MessageViewFragment` can be contained in the same screen and Activity; on a smaller screen (e.g., on a handset), each Fragment can be shown on a different screen and Activity.

4.3. Architectural design styles

The Android platform employs different types of components and connectors that enable a variety of architectural styles in Android apps. An architectural style is a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system (Taylor et al., 2009).

Our empirical investigation of Android apps, detailed later in Section 5, shows that Android supports five types of architectural styles: message-based explicit-invocation, message-based implicit-invocation, publish-subscribe, shared state, and distributed object styles. Table 1 summarizes the architectural styles observed in Android apps along with the elements involved in each style and the properties thereof.

As we discuss in the rest of this section, the architectural styles we observed in the Android framework promote several desirable properties in mobile systems. They facilitate software maintenance (**D2**), promote interoperability between third-party apps (**D3**), and allow evolvable and efficient mobile systems in the face of resource constraints (**D5**).

A pervasive architectural style in Android is the *message-based implicit-invocation* style, where components communicate using implicit Intent messages Google. Activities and Services are the types of components that participate in this style. This architectural style facilitates efficient, independent component operation since Intents can be handled asynchronously. This asynchronous communication allows components to be (1) offloaded as Android devices switch between apps and (2) resumed from their previous state, allowing components to be stopped and started in the face of resource constraints (**D5**). Loose coupling between components due to implicit invocation further enables Android components to be evolvable and maintainable (**D2**). Intent exchange, in this style, enables communication among heterogenous Android components (**D3**). As an example, `MessageCompose` of K-9 declares an Intent Filter for any Intent with the `SENDTO` action.

Another common style in Android is the *publish-subscribe* style, which is a specialization of the implicit-invocation style. For the publish-subscribe style in Android, Broadcast Receivers are leveraged to dispatch messages to multiple receiving components and receive Intents from other components, including the Android system itself. Activities and Services subscribe to particular Intents through the use of Intent Filters. These publish-subscribe mechanisms enable highly efficient dissemination of Intents. For K-9 in Fig. 2, the `Boot Receiver` dispatches system Intents—in this case, the Android system acts as a publisher—to the *`MailService`*, which acts as a subscriber.

In the *message-based explicit-invocation* style, components communicate directly using messages, i.e., explicit Intents in the case of Android Google. Message-based explicit-invocation represents the most commonly used style in Android, according to our results of mining the reverse-engineered architecture of hundreds of Android apps adopted from several repositories (cf. Section 5). Activities and Services are the Android components that can communicate using explicit-invocation style. This style retains the advantages of evolvability (**D2**) and handling of resource constraints present in the other message-based styles (**D5**); however, the message-based explicit-invocation style has the additional advantage of reducing communication overhead and unnecessary computation by targeting a specific component as a recipient of an Intent. For instance, in the example of Fig. 2, `MessageCompose` sends an explicit Intent to `EmailAddressList`.

The *shared state* style, where multiple components communicate through a shared data store, is realized in Android through Content Providers. These components provide a centralized, structured data store with uniform mechanisms for adding, removing, and querying stored data. Such components allow data to be accessed in an expedient manner and easily modified (**D2**). Separating data of an app into its own explicit type of component results in smaller components that have a lower memory footprint (**D5**). A data-access connector accessible through Android's `ContentResolver` class provides the interface to Content Providers. In Fig. 2 for K-9, `AttachmentProvider` enables file access to attachments; and `EmailProvider` allows access to messages of email accounts.

Finally, Android apps may conform to the *distributed objects* style, where objects in a distributed system communicate through RPC. For services that need to handle multi-threading and interprocess communication, an Android client (e.g., an Activity) and Service can leverage the Android Interface Definition Language (AIDL) to specify the interface through which they can communicate. The RPC mechanism allows developers to program in the familiar procedure-call paradigm (**D2**) while allowing interoperability between apps (**D3**). Similar to RPC mechanisms in CORBA (Vinoski, 1997), the AIDL specifies the interface of the Service that can be accessed; and a `Stub` class is created through which that interface can be accessed as if it were local. In Fig. 2, Activities `MessageList` and `Message Compose` leverage RPCs to request encryption services from the external app `OpenPGP`.

```
1  <uses-permission android:name="READ_CONTACTS"/>
2  <uses-permission android:name="INTERNET"/>
3  <uses-permission android:name="WRITE_EXTERNAL_STORAGE"/>
4  <activity
5   android:name=".activity.MessageCompose"
6   android:configChanges="locale"
7   android:enabled="false"
8   android:label="@string/app_name">
9   <intent-filter>
10  <action android:name="android.intent.action.SENDTO"/>
11  <data android:scheme="mailto"/>
12  <category android:name="android.intent.category.DEFAULT"/>
13  </intent-filter>
14 </activity>
15 <service
16  android:name=".service.RemoteControlService"
17  android:enabled="true"
18  android:permission="com.fsck.k9.permission.REMOTE_CONTROL"
19 />
```

Listing 1. Part of the Android manifest declaration for the K-9 mail app.

Table 2
Modeling evaluation rubric for the Android manifest.

Scope and purpose	Capturing the structure, configuration and interfaces of components in architectures that conform to the constraints imposed by the Android platform.
Basic elements	**Components** (in one of the pre-defined types of *Activity, Service*, and *Content Provider*), **interfaces** (called *Intent Filters*), **connectors** (called *Broadcast Receiver*), and **permissions** (defining both required and enforced permissions for each component)
Style	Manifest models conform to the Android event-driven architectural style; the constraints of that particular style are implicit in the model.
Static and dynamic aspects	Only static structure and interfaces are modeled.
Dynamic modeling	Although there is no direct support for dynamic modeling, Android establishes a close correspondence between the elements in the architectural model and the implementation constructs; thus, changes tracked in one can be straightforwardly applied to the other.
Non-functional aspects	Certain non-functional properties, such as required and enforced access control permissions and required hardware configurations, can be explicitly modeled.
Viewpoints	Structural viewpoint with certain points for capturing particular behavioral aspects of the app.

4.4. Software architecture models

Since the beginning of the 1990s, multiple ADLs have been developed to support software-architecture modeling, such as Rapide (Luckham and Vera, 1995), xADL (Dashofy et al., 2001), and Acme (Garlan et al., 2000). While each ADL is different and each one targets a particular domain or certain aspects of a system, the essential concepts common among such ADLs are explicit support for modeling key architectural elements: Components, Connectors, Interfaces, and Configurations (Medvidovic and Taylor, 2000). In turn, specification of an architecture in ADLs enables the analysis of the system for non-functional properties **(D5)**, most notably performance (Aquilani et al., 2001; Feiler et al., 2006), among others (Edwards et al., 2007).

These canonical architectural concepts are also identifiable in the modeling notations featured in the Android platform. Specifically, such architectural app descriptions are documented in a separate XML file, called *manifest*, that accompanies each app. An Android manifest has a canonical textual representation in which the app components and the required device features for the app, such as the minimum version of Android required and any hardware configurations required, are described. There is an associated form-based visualization that depicts the Android manifest configurations in a more accessible but less precise way.

Through the use of Android's manifest file, we are able to model the configuration of Android apps. Table 2 assesses the Android manifest with respect to the criteria suggested by Taylor et al. (2009) for evaluating software architecture modeling languages. The manifest file of apps installed on a device collectively comprise the description of overarching architecture in a mobile device. In the Android manifest, an app is modeled as a set of components. A component's declaration within the app's manifest can also include Intent Filters that specify the component's provided interfaces.

For example, Listing 1 shows part of the manifest for K-9, where two components, namely `MessageCompose` and `RemoteControlService`, are specified. Here, `MessageCompose`, which provides the ability to compose a new email, is associated with an Intent Filter (lines 9–13) that responds to `SENDTO` Intents. Other components (possibly in other apps) can then send an Intent with the `SENDTO` action to launch this Activity. Once the system has matched an Intent with `MessageCompose`'s Intent Filter, it launches the Activity and delivers the Intent.

There is no notion of explicit software connectors in manifest files, although a component of type Broadcast Receiver—that mainly facilitates system-wide message broadcasts through relaying messages to other components—could be interpreted as a connector. Configurations are not explicitly specified in the manifest, as bindings between interfaces are dynamically realized at runtime. This is similar to other dynamically-adaptable architectures and their descriptions (e.g., Taylor et al., 1996; Oreizy et al., 1998; 2008).

While the Android manifest modeling notations focus on capturing the functional aspects of a system—services provided by different components and connectors and the interactions that achieve the overall system functions—they also enable modeling some non-functional aspects, such as *required* and *enforced* per-

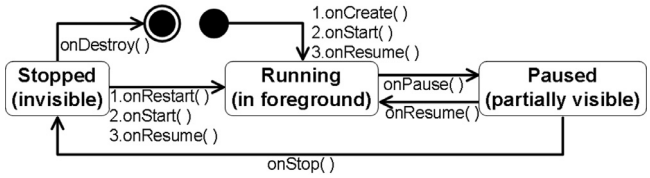

Fig. 3. Activity life-cycle methods.

missions that are Android's mechanisms for addressing the security concerns (**D4**). Required permissions are a set of permissions specifying the resources the app needs to run properly. An example of the required permissions shown in Listing 1 is `Internet` (line 2), which grants an app access to the Internet. Enforced permissions are a set of permissions that other apps must have in order to interact with this app. As an example, from Listing 1 we can see that `RemoteControlService` has specified interactions with this component require the `REMOTE_CONTROL` permission (line 18).

Permissions are the cornerstone for the Android security model (Enck et al., 2011; Felt et al., 2011). The permissions stated in the app manifest enable secure access to sensitive resources (**D5**) as well as cross-application interactions (**D3**). Separating specification of these permissions from the low-level implementations realized in the code provides both design and development conveniences (**D2**). It further facilitates the assessment of a system's security properties in terms of its configuration (**D4**) (Bagheri et al., 2015).

The Android manifest can be considered as a domain-specific ADL, optimized for describing architectures in a particular domain of the mobile-computing platform. Such a domain-specific modeling language results in several advantages. The language scope is particularly tailored to the needs of the Android app developers, rather than software developers in general. This scope, in turn, enables excluding unnecessary details and excessively verbose constructs, since the needs for generality are limited. Moreover, Android domain knowledge is directly encoded into the manifest language semantics rather than being repeated in every model. As a concrete example, since the event-driven style mandates the use of an event-bus connector—realized by the Android framework—between each pair of linked components, there is no need to include the notion of such a connector in the manifest language, as the developers and the tool set simply assume the existence of such a connector on each link implicitly. However, the ADL-like manifest of an Android app does not thoroughly support the specification of all Android-specific architectural concepts, most notably the required interfaces of Android components. We discuss these deficiencies in more details in Section 6.

4.5. Architecture implementation and deployment

A particular challenge in the realization of software architectures is the gap between the architectural elements, such as components and connectors, and their implementation in terms of programming-language constructs, such as classes, pointers, and variables (Bagheri and Sullivan, 2012). Lack of systematic support for mapping an architecture to its implementation can result in architectural drift (Taylor et al., 2009), a situation where the architectural models and their implementation do not match.

To address this challenge, software architecture-research advocated the development of *architecture implementation frameworks*—a piece of software that acts as a bridge between a particular architecture style and a set of implementation technologies. It provides key elements of the architectural style in code, in a way that assists developers in implementing systems that conform to the prescriptions and constraints of the style (Taylor et al., 2009; Bass et al., 2003). Examples of architectural implementation frameworks include ArchJava (Aldrich et al., 2002), which provided support for realizing architectures by extending Java with new programming-language constructs, and C2 framework (Taylor et al., 1996), which provided a class library for realizing architectures in the C2 style. Software-architecture research also paved the way for the application of architecture-based development in the mobile setting. For instance, Prism-MW (Malek et al., 2005) was the first middleware for the architecture-based development of mobile software in Java.

Interestingly, our empirical investigation, detailed in Section 5, reveals that these principles, first conceptualized in software-architecture research, have also been realized in the Android framework. While Android's architectural concepts and designs outlined in Section 4.1 provide a certain level of uniformity among third-party apps, they are not sufficient for ensuring the actual implementations of those apps abide by the Android-specific architectural constraints and rules. Without such assurances, it is neither possible to achieve a robust Android ecosystem (**D1**) nor interoperability (**D3**). At the same time, without proper development support, the developer's productivity (**D2**) would suffer, as each developer would be forced to reinvent an implementation that realizes Android's architectural concepts. To that end, Android has codified architectural knowledge in two ways: an *Android Development Framework (ADF)* and an *Android Runtime Environment (ART)*.

Similar to prior architecture implementation frameworks, ADF provides a library of classes to help the developers implement their apps. Specifically, ADF provides classes that developers extend to realize application-specific logic. For instance, the Activity component is realized by extending ADF's Activity class and providing an implementation for its various life-cycle methods. Fig. 3 shows the Activity's life-cycle methods that the developer needs to implement. The methods correspond to the callbacks an app receives from the Android system. For example, *onCreate()* method is called when the app is instantiated Google.

On the other hand, ART provides the runtime facilities for realizing Android's family of architectures. In particular, ART provides the implementation of Android connectors, such as message-based, RPC, and data access connectors (recall Section 4.1). They correspond to the communication facilities in Android that are application-independent and can be reused in the construction of any app. By separating the application-specific logic from the reusable communication logic, Android also facilitates interoperability among third-party apps (**D3**).

In addition, the Android architecture plays a significant role in the secure development of software systems (**D4**), through isolating apps from each other and system resources from apps via a sandboxing mechanism Google. Each app runs in its own virtual machine, and can only access its own files by default, protecting apps with sensitive information from others. In addition to providing inter-app isolation, Android requires app interactions to occur through certain, well-defined interfaces. This design, in turn, allows developers to narrow the scope of a system's security analysis to those interfaces (Chin et al., 2011).

4.6. Architectural support for non-functional properties

In a resource-constrained environment, such as mobile platforms, efficiency becomes a non-functional property of utmost importance (Hao et al., 2013). Software architecture can significantly improve the prospects for achieving efficiency goals (Taylor et al., 2009). In fact, different architectural elements (i.e., components, connectors, etc.) and their configurations have a direct and critical impact on the system's efficiency. In addition to such design choices, architectural tactics, defined by Bachman as *"means*

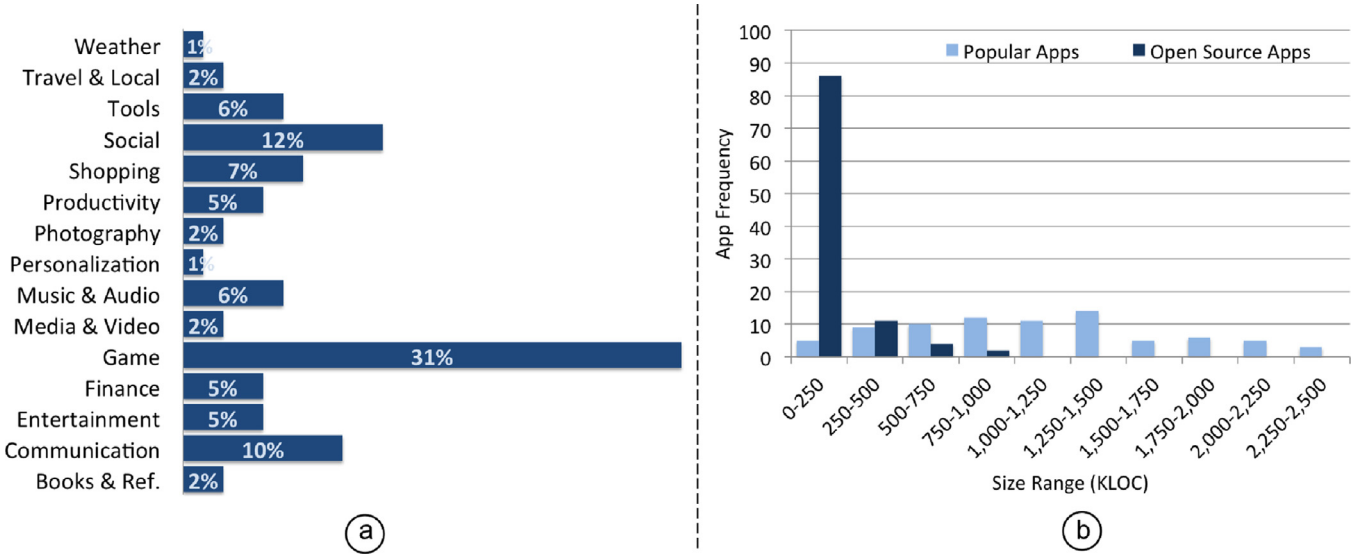

Fig. 4. Distribution of selected apps by (a) category, and (b) size.

of satisfying a quality-attribute-response measure" (Bachmann et al., 2003), have a significant impact on a system's non-functional properties.

Starting from the components, the rule of thumb is to *keep components small*, meaning that each component should ideally perform a single, specific task in the system (Taylor et al., 2009). In Android, different types of components are defined for different purposes, and each one is dedicated to a limited-scoped task. For instance, recall from Section 4.1 that Activity components provide an app's user interfaces, and each Activity component is scoped to represent a particular screen of the app. The Android system runs only one Activity at a time, which reduces memory utilization and improves efficiency (**D5**). By decomposing the software into its architectural constructs, Android is able to offload parts of the software that are not needed at runtime. Android realizes this through the notion of *component life-cycle* (recall Section 4.5), e.g., apps need to implement *onPause* and *onResume* callbacks, which are called when the user switches between apps Google.

Fragments also have implications on resource constraints, especially on performance and energy consumption Google. To prevent redrawing of an entire screen if only a portion of it changes, a developer can utilize a Fragment to modularize the screen into regions that can change independently. As an example from K-9, if a new message arrives, only the part of the screen corresponding to `MessageListFragment` would need to be redrawn, while the `MessageViewFragment` portion of the screen would remain unchanged. This results in increased performance and decreased energy consumption (**D5**), due to the reduced need to redraw the screen.

Connectors facilitate interactions among components, and thus can have a direct impact on the system's performance (Taylor et al., 2009). The Broadcast Receiver in Android, which as alluded to earlier behaves as a connector, is available in different flavors; careful selection among them is thus important. For instance, Android supports three types of broadcast Intents Google: (1) *Normal broadcasts* are dispatched to all registered Broadcast Receivers immediately. This type of broadcast is typically the least efficient since it potentially executes all registered Broadcast Receivers. (2)*Ordered broadcasts* are sent to one Broadcast Receiver component at a time; and Broadcast Receiver components can set their priority level for receiving ordered broadcasts. The Intent propagation can be halted by any Broadcast Receiver in the delivery chain of an ordered broadcast Intent. Using ordered broadcast, higher priority Broadcast Receivers in the delivery chain can prevent unnecessary propagation, resulting in efficiency gains. Finally, (3) *sticky broadcasts* remain in the system for re-broadcasting to future Broadcast Receivers after they have been delivered. This form of broadcast is a form of caching, which can be utilized to improve performance.

The Broadcast Receiver can also be modified through the *Alarm Manager* Google, a scheduler that, among other things, handles access to the typically energy-greedy system APIs. One particular scheduling mechanism that can be used by the Alarm Manager to improve efficiency is *InexactRepeating*. The key idea here is to reduce the number of times the system needs to access power-consuming functionality, such as GPS, through synchronizing requests from various apps. These scheduling mechanisms provide arbitration facilities to modify the behavior of the Broadcast Receiver (Mehta et al., 2000), and in turn to make the Intent dissemination between the system's components more efficient (**D5**).

5. Architectural characteristics of Android ecosystem

To further understand the architectural characteristics of Android, we determined the extent to which Android utilizes styles, components, and connectors. To that end, we collected over 1400 apps from four different app repositories: 100 randomly selected apps and an additional non-overlapping set containing the 100 most popular free apps (200 in total) from Google play, the official Android repository; all 1046 available apps (circa August 2014) from F-droid, a repository with only open-source apps; 100 popular apps from Bazaar, a third-party Android market from Iran; and another 100 apps from Wandoujia, one of China's largest app markets.

Fig. 4 (a) represents the distribution of apps used in our experiments from different categories, showing that they are sufficiently diverse, and representative of what one can find installed on a typical device. According to the diagram, Game, Social, and Communication are the categories that have the highest numbers of apps in our data set, in that order. Fig. 4 shows the distribution of selected apps by their size. Interpretation of data shows that popular apps are distributed normally, with the mode of 1250–1500 KLOC. The majority of open-source apps are, however, small-sized apps.

We analyzed each app's configuration file and its source/bytecode to extract the app's architectural characteristics. For this purpose, we leveraged apktool Android apktool to decode the app's

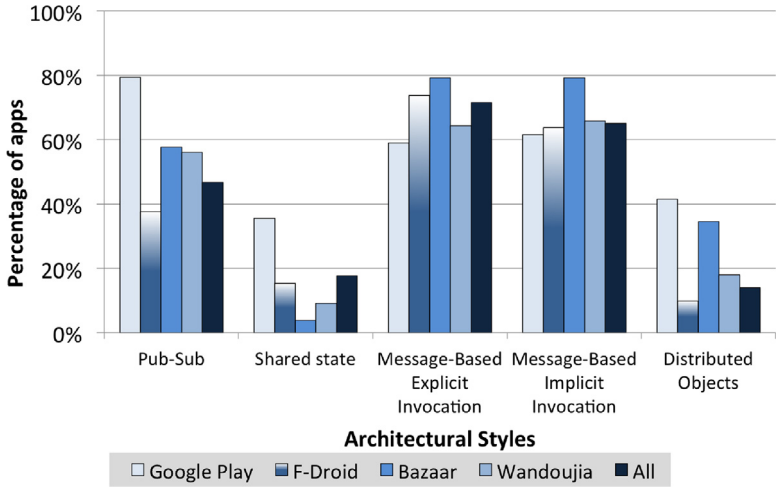

Fig. 5. Percentage of architectural styles usages in apps of four repositories.

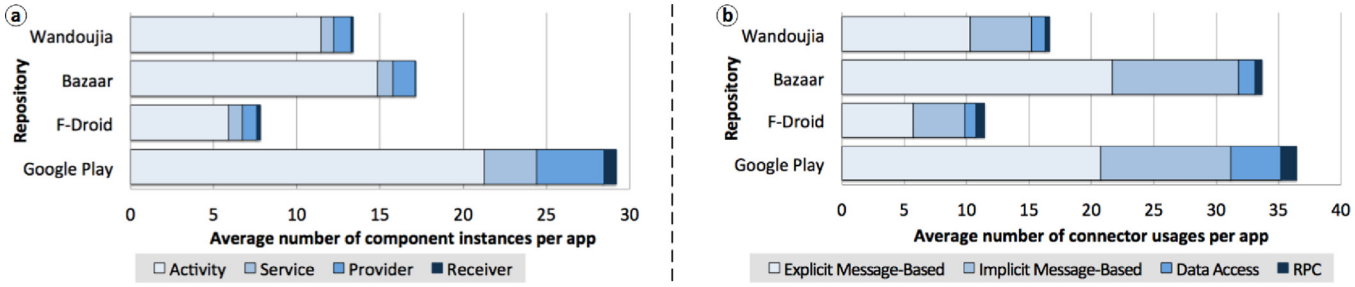

Fig. 6. Average number of Android (a) component instances, and (b) connector usages in each app of four repositories.

manifest file, and Valle é-Rai et al. (1999) to analyze its code and extract additional information not available in the manifest file. Our architectural recovery approach is quite straightforward, as it simply requires tracking the architectural building blocks defined in Section 4.1 in the app's implementation. Fig. 5 depicts all of the identified styles (recall Table 1) in the four app repositories. Overall, all of the styles we identified are used by every repository with the three different variants of message-based styles (including publish-subscribe) being the most dominant: Each variant of a message-based style is used on average for all repositories by more than 45% of apps. The message-based implicit and explicit invocation styles are the most used styles in all apps across all repositories, making Intents the dominant communication mechanism for Android apps.

Google Play apps, which are the most widely used set of apps, take advantage of a variety of styles more often than apps from other repositories: each style is leveraged by at least 35% of Google Play apps and by as many as 80% of Google Play apps. In contrast, across all repositories, each style is leveraged by at least 15% of all apps and by as many as 72% of all apps.

Figures 6 a and b show the average number of component instances and connector usages in each app for different repositories. Overall, Android app architectures make use of a wide variety of components and connectors, which we argue has aided Android's success. On average, an Android app makes use of between 7–27 components and 12–36 connectors, depending on the repository it originates from. Google Play apps, the most widely used group of apps, have the highest number of components and connectors (29 components, and 37 connectors per app). In all four repositories, the ranking of components in frequency from highest to lowest is: Activity, Content Provider, Service, and Broadcast Receiver. This ranking indicates that user-interface functionality and data storage are the most dominant concerns of Android app architectures.

Table 3
Average number of components and their provided ports in each popular app from the Google Play store for different categories.

Category	Average number in each app				
	Components				Provided
	Activity	Service	Receiver	Provider	Ports
Books & reference	57.50	6.50	5.50	2.00	18.00
Communication	61.38	7.38	12.00	2.38	44.38
Entertainment	21.40	3.60	3.40	1.20	11.40
Finance	111.50	7.50	4.75	0.75	17.00
Game	17.88	1.76	2.92	0.12	4.48
Media & video	19.00	4.50	4.50	0.50	20.00
Music & audio	22.80	5.20	7.20	1.60	20.80
Personalization	37.00	6.00	8.00	3.00	18.00
Photography	8.65	1.43	2.25	0.29	5.21
Productivity	30.00	3.75	3.25	3.25	20.00
Shopping	50.50	6.67	6.17	1.83	16.33
Social	51.20	11.40	7.20	3.50	24.20
Tools	32.40	6.20	5.20	0.00	12.60
Travel & local	8.50	1.43	2.24	0.29	5.19
Weather	17.00	11.00	11.00	0.00	20.00

Table 3 shows the average number of component instances and provided ports in each popular app from the Google Play store across different categories. *Finance* apps, such as banking or payment systems, provide richer user-interface compared to the other kinds of apps, as they have the highest number of Activity components (on average 111.50 Activity components per app). Running background tasks (via Service components) as well as representing and manipulating data (via Content Provider components) are

largely observable in *Social* apps. Interpretation of data also reveals that *Communication* apps, such as messaging and telephony apps, that largely depend on listening, receiving, and handling systems events, have the highest number of Receiver components and provided ports.

There are several conclusions we can draw from this data. First, the results show that Android apps indeed have quite complex and interesting architectures, making use of the whole gamut of architectural elements found in Android. Since there is generally a widespread misconception that apps are simple extensions of a platform, we believe these results may be surprising to some. While apps may have been quite small early in the evolution of Android, our study shows that today's apps consist of tens of components and connectors, configured to make use of multiple styles; thus, they are not that different from architectures one may find in traditional desktop software. Second, this study, consistent with other prior studies Bagheri et al. (2015); Chin et al. (2011), shows that it is possible to recover very precise models of the system's architecture through straightforward program-analysis techniques, due to the existence of architectural elements as essential building block of Android systems (cf. Section 4.1). This should be of interest to the software-architecture research community that for years has struggled with the development of accurate architectural recovery techniques to study open-source software and to test hypotheses Garcia et al. (2013). Third, since in the Android domain architectural recovery can be achieved accurately, it is significantly easier to check the conformance of an app's implementation against its prescriptive architecture, thereby reducing the possibility of architectural erosion.

6. Discussion

Although Android's app architecture leverages a number of advantageous design decisions, several aspects of that architecture deviate from principles advocated in the literature, in some cases resulting in deficiencies and limitations that deserve attention. In this section, we attempt to shed light on some of these Android-specific architectural smells and anti-patterns found in our study.

6.1. Inconsistent hierarchical (de)composition

In Section 4.2, we described the three levels of hierarchical (de)composition possible in Android: (1) at the highest-level of granularity, each mobile device is comprised of an architecture, where its computational elements are the apps, possibly interacting with one another through one of Android's inter-app communication mechanisms, (2) each app, in turn, has an architectural configuration, consisting of the four component types and four connector types discussed in Section 4.1, and (3) at the lowest level of granularity, Android's Activity components may have an internal architecture, consisting of Fragment components.

Although Android's hierarchies provide plenty of benefits, they have several shortcomings in the way they have been realized. In particular, software-architecture research has advocated for recursive rules of (de)composition, whereby a component can be decomposed into lower-level components of the same type and vice versa (Taylor et al., 2009). One benefit of such an approach, experienced in architectural frameworks, such as Prism-MW (Malek et al., 2005), as well as composite services in SOA (Malek et al., 2005; Erl, 2006), is that the same set of composition rules can be applied at different levels of hierarchy.

On the other hand, Android provides constructs with arbitrarily different structures, behaviors, and semantics at different levels of hierarchy. Similarly, the rules of composition at one level are completely different from another level. For instance, the way Fragments are composed when constructing an Activity is completely different from the manner in which the four component types are composed when constructing an app. As another example, consider that while a manifest file is used to model the configuration of an app's architecture, a similar concept is not made available for use at the highest and lowest levels of granularity. These discrepancies between different levels of hierarchy are at best an annoyance, but possibly a hindrance to a developer's ability to realize architectures that are better suited for a system.

6.2. Extensibility limitations

In Section 4.1, we described that Android constrains components to four specific types (i.e., Activities, Services, Content Providers, and Broadcast receivers) in order to achieve the provision of more uniform user interfaces, interoperability between components, and a separation of concerns based on functionality common to mobile systems (e.g., background services, data stores, and UI handling). However, Android provides limited support for building other component types that do not fit the semantics of Android's pre-defined components. For example, an Android game, a category of apps popular in mobile-computing ecosystems, would commonly include a Game Engine—but none of the four predefined component types provide a good match, since such an engine should manage other components in the app and may implement its own unique life-cycle.

We believe a generic Android component, similar to those suggested in prior architecture implementation frameworks (Malek et al., 2007), would enable developers to extend the framework with custom-built component types, while still maintaining properties that are important for mobile systems (e.g., efficiency and loose coupling).

Connectors in Android face similar extensibility limitations. Notably, they lack the ability to build configurations corresponding to certain topologies. For instance, Android lacks support for isolating groups of components into their own layers (e.g., a layer for game-logic components and another layer for graphics-rendering components), where each layer is separated by a different event bus. Such a design provides loose coupling and separation of concerns at a granularity above Android components.

6.3. Omission of architectural concepts

In Section 4.4, we described that Android allows specification of built-in components and provided interfaces (i.e., Intent Filters) in an app's source code or through the ADL-like manifest. However, the ADL-like manifest of an Android app does not allow the specification of other key elements that are typically found in traditional ADLs: (1) connectors, (2) configurations resulting from interconnections between components and connectors, and (3) required interfaces of components or connectors. The omission of these architectural concepts in Android apps hinder architectural analysis and understandability of an app.

A particular selection of connectors in an app and the way in which those connectors and the app's components are interconnected (i.e., the app's architectural configuration) imply different non-functional properties (e.g., high or low efficiency or maintainability). The inability to easily obtain such information from a single source makes analyzing an Android app for a non-functional property (e.g., security), or conducting automated architectural conformance checks, significantly more challenging. More specifically, an engineer would need to either manually read the code or utilize a static or dynamic analysis to obtain such information, either of which would be cumbersome, time-consuming, or error prone.

Although Android supports specification of provided interfaces through Intent Filters, no analogous mechanism exists for required

interfaces. To determine the required interfaces of an Android component, Intents for implicit invocation, Content Resolvers for data access, and Android RPC mechanisms must be analyzed (e.g., through static analysis). Consequently, analysis of non-functional properties (e.g., security or maintainability) or performing architectural conformance are hindered by the hidden required interfaces of components, especially given the lack of mechanisms for automatically determining data accesses and RPCs in Android. To obtain this information, a developer may need to read code, eliminating the advantages of reducing complexity through focusing on architectural abstractions.

6.4. Broadcast receiver's connector envy

Broadcast Receivers poorly separate concerns, resulting in deficiencies that affect maintainability and efficiency (cf. Section 4.1). This poor separation of concerns stems from the fact that Broadcast Receivers tend to include application-specific logic, which are the responsibility of components, and distribute Intents to other components, an application-independent concern handled by connectors.

We believe that the Broadcast Receiver is best to be clearly defined as a connector, discouraging developers of including application-specific logic within such a module that by design should behave as a gateway to other components, a responsibility advocated to assign to connectors in the architecture literature (Mehta et al., 2000). Essentially, the Broadcast Receiver in Android by itself can be a complicated building block, supporting three different types of broadcast Intents, that incorporating application-specific logic into may make it very hard to understand. In fact, the phenomenon, where a component acts also as a connector, is referred to as *Connector Envy* (Garcia et al., 2009), an architectural-decay instance that results in bloated components, reduces efficiency by increasing each affected component's memory footprint, and renders such components more complex and less maintainable.

6.5. Model-view-controller anti-pattern

Another poor separation of concerns occurs due to Android's design that breaks the Model-View-Controller (MVC) architectural pattern (Krasner et al., 1988; Taylor et al., 2009). In that pattern, a View component provides graphical depictions of information; a Model component contains information to be depicted; and a Controller component maintains consistency between the Model and View. In Android, a Content Provider acts as a Model; the responsibilities of the View and Controller are integrated into one component, an Activity (cf. Section 4.1). This violates the single-responsibility design principle (Gamma et al., 1994), resulting in a poor separation of concerns that breaks the MVC pattern (Sokolova et al., 2013). This design decision results in an architectural anti-pattern (Brown et al., 1998), which is a recurring set of architectural elements that together negatively affects the architecture of a system, and renders the system more complex and less maintainable. For example, testing the operations of the View and Controller in Android cannot be done separately.

We recommend separating the responsibilities of event handling and interface management into distinct components, on the basis of the Android-specific MVC architecture proposed by Sokolova et al. (2013). Architectural support for such a separation of concerns at the framework-level not only facilitates the development of less complex components and well-structured applications, but also discourages development of "bloated" Activity components that suffer from increased memory footprints and decreased efficiency.

7. Related work

Software architecture recovery techniques have been around for over three decades (Ducasse and Pollet, 2009; Koschke, Springer Berlin Heidelberg, 2009; Maqbool and Babri, 2007; Shtern and Tzerpos, 2012). A majority of such techniques group implementation-level entities (e.g., files, classes, or functions) into clusters, where each cluster represents an architectural component (Fiutem et al., 2002; Guo et al., 1999; Tzerpos and Holt, 2000; Sartipi, 2001; 2003; Fiutem et al., 1999). Among others, Guo et al. (1999) presented the Software Architecture Reconstruction Method (ARM), which is used to specify design patterns; the focus of ARM, similar to many other techniques we studied, is on object-oriented programming language constructs. Such constructs are typically not considered architectural. Although they discussed the possibility of ARM for identification of higher-level architectural patterns, how this would actually be accomplished is not explained.

Different from these research efforts that extract canonical architectural notions from low-level implementation entities, this work is geared towards the application of architectural recovery techniques in the context of Android ecosystem, where a common framework underlying the ecosystem of apps provides extensive support for architectural concepts. In fact, by tracking the architectural building blocks provided by the framework and extended in individual apps, this work studies the extent to which the Android's ecosystem of apps employs architectural concepts.

Ali and Solis (2014) proposed the use of architectural models at runtime to support monitoring the adaptation, evolution, and maintenance of mobile software systems. Kim (2013) also proposed a reference architecture template for developing adaptive mobile apps. While the focus of these research efforts is on architectural support for software adaptation in diverse mobile platforms, the specific characteristics of Android platform has not been investigated in detail. They do not analyze, among other things, architectural characteristics featured in the Android framework, nor the extent to which real-world apps actually leverage such framework-provided, architectural constructs.

The other relevant line of research focuses on applications of recovered architecture in a variety of mobile software engineering problems (Bagheri et al., 2015; 2016; Mirzaei et al., 2016; Schmerl et al., 2015). Among others, COVERT (Bagheri et al., 2015) showed the power of software architectural abstractions for the analysis of security properties in Android apps. It relies on both static analysis and formal model checking, backed with the definition of an architecture style for analyzing security properties of Android apps. Along the same line, Schmerl et al. (2015); 2016) proposed an architecture-based approach for run-time mitigation of Android Intent vulnerabilities. Joorabchi and Mesbah (2012) presented a reverse engineering technique to automatically navigate a given iPhone app, and to recover its architectural model at the GUI level. Mirzaei et al. (2016) used the recovered architecture of apps, mainly at the GUI-level, for combinatorial, yet scalable, GUI testing of Android apps. TrimDroid leveraged the dependencies among the architectural elements comprising a given app to reduce the number of combinations in generated test cases.

We share with these research efforts the common insight on using software architecture as a crucial means that provides a proper level of abstraction for representing issues and modeling problems. Different from all prior research efforts, this work focuses on studying the extent to which Android apps employ architectural concepts in practice. To the best of our knowledge, this study is the most comprehensive and elaborate investigation of the architectural characteristics of Android ecosystem, supported with mining the reverse-engineered architecture of hundreds of Android apps in several app repositories.

8. Conclusion

This paper described the role of software architecture in the design and construction of modern mobile software. Specifically, we identified several mobile-computing drivers that motivated the adoption of architectural principles, distilled those principles, and traced them back to their conception in the literature. By mining the reverse-engineered architecture of hundreds of apps, we found that Android apps are complex, consisting of tens of components and connectors, and often make use of multiple styles. We also described several Android-specific architectural-decays and anti-patterns that were identified in our study.

Since we are not aware of any other work that has conducted a similar in-depth analysis of Android's app architecture, we believe our work could have a significant impact on both practitioners and researchers, as follows. The architectural concepts described throughout our study could help practitioners understand the implications of their design decisions. We found many improper usages of architectural constructs and styles in the reverse-engineered architecture of popular apps. For instance, in cases where developers had used the message-based explicit-invocation style to send an Intent to multiple components, it would have been significantly more efficient and architecturally elegant to use the publish-subscribe style for such group communication. Such architectural mistakes are due to the lack of a deeper understanding of how design choices may affect the system's properties. We believe that our study also has the potential to impact the research community by demonstrating the rich body of architectural knowledge that can be readily mined from app ecosystems, as well as shedding light on the architectural principles that were adopted and those that were deviated from in Android. In turn, we expect this to set the stage for further research in the future.

Acknowledgment

This work was supported in part by awards CCF-1252644 from the National Science Foundation, D11AP00282 from the Defense Advanced Research Projects Agency, W911NF-09-1-0273 from the Army Research Office, HSHQDC-14-C-B0040 from the Department of Homeland Security, and FA95501610030 from the Air Force Office of Scientific Research.

References

Aldrich, J., Chambers, C., Notkin, D., 2002. Archjava: connecting software architecture to implementation. In: ICSE, pp. 187–197.
Ali, N., Solis, C., 2014. Mobile architectures at runtime: research challenges. In: Proceedings of MOOBILESoft.
Android apktool. https://code.google.com/p/android-apktool/.
Aquilani, F., Balsamo, S., Inverardi, P., 2001. Performance analysis at the software architectural design level. Perform. Eval. 45 (23), 147–178. doi:10.1016/S0166-5316(01)00035-9. http://www.sciencedirect.com/science/article/pii/S0166531601000359.
Bachmann, F., Bass, L., Klein, M., 2003. Deriving architectural tactics: a step toward methodical architectural design. Technical Report.
Bagheri, H., Sadeghi, A., Garcia, J., Malek, S., 2015. Covert: compositional analysis of android inter-app permission leakage. IEEE Trans. Software Eng. 41 (9), 866–886.
Bagheri, H., Sadeghi, A., Jabbarvand, R., Malek, S., 2016. Practical, formal synthesis and automatic enforcement of security policies for android. In: Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).
Bagheri, H., Sullivan, K., 2012. Pol: specification-driven synthesis of architectural code frameworks for platform-based applications. In: Proceedings of the 11th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE'12). ACM, Dresden, Germany, pp. 93–102.
Bass, L., Clements, P., Kazman, R., 2003. Software Architectures in Practice. Addison Wesley.
Bazaar. https://cafebazaar.ir/.
Bosch, J., 2009. From software product lines to software ecosystems. In: International Software Product Line Conference,San Francisco, California, pp. 111–119.
Brown, W.J., McCormick, H.W., Mowbray, T.J., Malveau, R.C., 1998. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, New York.
Chin, E., Felt, A.P., Greenwood, K., Wagner, D., 2011. Analyzing inter-application communication in android. In: International Conference on Mobile Systems, Applications, and Services,New York, NY, USA, pp. 239–252.
Dashofy, E.M., Hoek, A.V.d., Taylor, R.N., 2001. A highly-extensible, xml-based architecture description language. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture. IEEE Comput. Society, Washington, DC, USA, p. 103. doi:10.1109/WICSA.2001.948416. http://dx.doi.org/10.1109/WICSA.2001.948416.
Ducasse, S., Pollet, D., 2009. Software architecture reconstruction: a process-oriented taxonomy. IEEE Trans. Softw. Eng. 35 (4), 573–591.
Ebert, C., Jones, C., 2009. Architecture for embedded open software ecosystems. Comput. 42, 42–52. http://dx.doi.org/10.1109/MC.2009.118.
Edwards, G., Malek, S., Medvidovic, N., 2007. Scenario-driven dynamic analysis of distributed architectures. In: Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering. Springer-Verlag, Berlin, Heidelberg, pp. 125–139. http://dl.acm.org/citation.cfm?id=1759394.1759411.
Eklund, U., Bosch, J., 2014. Architecture for embedded open software ecosystems. J. Syst. Softw. 92, 128–142. doi:10.1016/j.jss.2014.01.009. http://www.sciencedirect.com/science/article/pii/S0164121214000211.
Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S., 2011. A study of android application security. In: Proceedings of the 20th USENIX Conference on Security. USENIX Association, San Francisco, CA. p. 21.
Erl, T., 2006. Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall, Upper Saddle River, NJ, USA.
Feiler, P.H., Lewis, B.A., Vestal, S., 2006. The SAE architecture analysis & design language (AADL) a standard for engineering performance critical systems. IEEE, pp. 1206–1211. doi:10.1109/CACSD-CCA-ISIC.2006.4776814. http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4776814.
Felt, A.P., Hanna, S., Chin, E., Wang, H.J., Moshchuk, E., 2011. Permission re-delegation: attacks and defenses. In 20th Usenix Security Symposium, San Francisco, CA.
Fiutem, R., Antoniol, G., Tonella, P., Merlo, E., 1999. Art: an architectural reverse engineering environment. J. Softw. Maintenance 11 (5), 339–364.
Fiutem, R., Tonella, P., Anteniol, G., Merlo, E., 2002. A cliche-based environment to support architectural reverse engineering. In: Proceedings of the third Working Conference on Reverse Engineering (WCRE), pp. 277–286.
F-droid. https://f-droid.org/.
Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison- Wesley Professional.
Garcia, J., Ivkovic, I., Medvidovic, N., 2013. A comparative analysis of software architecture recovery techniques. In: International Conference on Automated Software Engineering, Palo Alto, CA, pp. 486–496.
Garcia, J., Popescu, D., Safi, G., Halfond, W., Nenad, M., 2009. Identifying architectural bad smells. In: European Conference on Software Maintenance and Reengineering, pp. 255–258.
Garlan, D., Monroe, R.T., Wile, D., 2000. Acme: architectural description of component-based systems. Foundations of Component-Based Systems.
Google, Android activity component. http://developer.android.com/guide/components/activities.html.
Google, Android alarm manager. http://developer.android.com/intl/ru/reference/android/app/AlarmManager.html.
Google, Android application fundamentals. http://developer.android.com/guide/components/fundamentals.html.
Google, Android broadcast receiver component. http://developer.android.com/reference/android/content/BroadcastReceiver.html.
Google, Android content provider component. http://developer.android.com/guide/topics/providers/content-providers.html.
Google, Android fragment. http://developer.android.com/reference/android/app/Fragment.html.
Google, Android intent. http://developer.android.com/reference/android/content/Intent.html.
Google, Android intent filters. http://developer.android.com/guide/topics/intents/intents-filters.html.
Google, Android interface definition language (aidl). http://developer.android.com/guide/components/aidl.html.
Google, Android service component. http://developer.android.com/guide/components/services.html.
Google, Android system permissions. http://developer.android.com/guide/topics/security/permissions.html.
Guo, G., Atlee, J., Kazman, R., 1999. A software architecture reconstruction method. In: Proceedings of the third Working International Conference on Software Architecture (WICSA).
Google play market. http://play.google.com/store/apps/.
Hao, S., Li, D., Halfond, W.G., Govindan, R., 2013. Estimating mobile application energy consumption using program analysis. In: The International Conference on Software Engineering.
Joorabchi, M.E., Mesbah, A., 2012. Reverse engineering ios mobile applications. In: Proceedings of the third Working Conference on Reverse Engineering (WCRE).
Kim, H.-K., 2013. Architecture for adaptive mobile applications. Int. J. Bio-Sci. Bio-Technol. 5 (5), 197–210. doi:10.14257/ijbsbt.2013.5.5.21. http://www.sersc.org/journals/IJBSBT/vol5_no5/21.pdf.
Koschke, R., 2009. Architecture reconstruction. In: Software Engineering. Springer, Berlin Heidelberg, pp. 140–173.
Krasner, G.E., Pope, S.T., et al., 1988. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. J. Object Oriented Program. 1 (3), 26–49.

Kruchten, P.B., 1995. The 4+1 view model of architecture. IEEE Softw. 12 (6), 42–50.
La Polla, M., Martinelli, F., Sgandurra, D., 2013. A survey on security for mobile devices. Commun. Surv. Tut., IEEE 15 (1), 446–471.
Luckham, D.C., Vera, J., 1995. An event-based architecture definition language. IEEE Trans. Softw. Eng. 21 (9), 717–734. doi:10.1109/32.464548.
Malek, S., Mikic-Rakic, M., Medvidovic, N., 2005. A style-aware architectural middleware for resource-constrained, distributed systems. IEEE Trans. Softw. Eng. 31 (3), 256–272.
Malek, S., Mikic-Rakic, M., Medvidovic, N., 2007. Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. In: ICSE. IEEE Computer Society, Washington, DC, USA, pp. 591–601.
Manikas, K., Hansen, K.M., 2013. Software ecosystems - a systematic literature review. J. Syst. Softw. 86 (5), 1294–1306.
Maqbool, O., Babri, H., 2007. Hierarchical clustering for software architecture recovery. IEEE Trans. Softw. Eng. 33 (11), 759–780.
Medvidovic, N., Mikic-Rakic, M., Mehta, N.R., Malek, S., 2003. Software architectural support for handheld computing. IEEE Comput. 36 (9), 66–73.
Medvidovic, N., Taylor, R., 2000. A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. 26 (1), 70–93.
Mehta, N.R., Medvidovic, N., Phadke, S., 2000. Towards a taxonomy of software connectors. In: ICSE. ACM, pp. 178–187.
Mirzaei, N., Garcia, J., Bagheri, H., Sadeghi, A., Malek, S., 2016. Reducing combinatorics in gui testing of android applications. In: Proceedings of ICSE.
Nikzad, N., Chipara, O., Griswold, W.G., 2014. Ape: An annotation language and middleware for energy-efficient mobile application development. In: ICSE, pp. 591–601.
Oreizy, P., Medvidovic, N., Taylor, R.N., 1998. Architecture-based runtime software evolution. In: Proceedings of the 20th International Conference on Software Engineering. IEEE Computer Society, Kyoto, Japan, pp. 177–186. http://portal.acm.org/citation.cfm?id=302181.
Oreizy, P., Medvidovic, N., Taylor, R.N., 2008. Runtime software adaptation: framework, approaches, and styles. In: Companion of the 30th International Conference on Software Engineering. IEEE Comput. Soc., pp. 899–910.
Picco, G.P., Julien, C., Murphy, A.L., Musolesi, M., Roman, G.-C., 2014. Software engineering for mobility: reflecting on the past, peering into the future. In: 2014 Future of Software Engineering (FOSE'14), pp. 13–28.
Sartipi, K., 2001. Alborz: a query-based tool for software architecture recovery. In: Proceedings of the International Workshop on Program Comprehension (IWPC).
Sartipi, K., 2003. Software architecture recovery based on pattern matching. In: Proceedings of the International Conference on Software Maintenance (ICSM).
Schmerl, B., Gennari, J., Camara, J., Garlan, D., 2016. Raindroid – a system for run–time mitigation of android intent vulnerabilities: Poster. In: Proceedings of the 2016 Symposium and Bootcamp on the Science of Security *(HotSoS'16)*.
Schmerl, B., Gennari, J., Garlan., D., 2015. An architecture style for android security analysis: poster. In: Proceedings of the 2015 Symposium and Bootcamp on the Science of Security *(HotSoS'15)*, pp. 15:1–15:2.
Shaw, M., Garlan, D., 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
Shtern, M., Tzerpos, V., 2012. Clustering methodologies for software engineering. Adv. Softw. Eng.
Sokolova, K., Lemercier, M., Garcia, L., 2013. Android passive mvc: a novel architecture model for android application development. In: International Conference on Pervasive Patterns and Applications.
Taylor, R., Medivodivc, N., Dashofy, E., 2009. Software Architecture: Foundations, Theory, and Practice. Wiley, Hoboken, NJ, USA.
Taylor, R., Medvidovic, N., Anderson, K.M., James Whitehead, J.E., Robbins, J.E., 1996. A component- and message-based architectural style for gui software. IEEE Trans. Softw. Eng. 22 (6), 390–406.
Tzerpos, V., Holt, R., 2000. Acdc: an algorithm for comprehension-driven clustering. In: Proceedings of the Working Conference on Reverse Engineering (WCRE).
Valle é-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V., 1999. Soot - a java bytecode optimization framework. In: Conference of the Centre for Advanced Studies on Collaborative research, CASCON'99.
Vinoski, S., 1997. Corba: integrating diverse applications within distributed heterogeneous environments. IEEE Commun. Mag. 35 (2), 46–55.
Wasserman, A., 2010. Software engineering issues for mobile application development. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER'10), pp. 397–400.
Wandoujia. https://wandoujia.com/.
Wolf, A., Perry, D., 1992. Foundations for the study of software architecture. ACM SIGSOFT Softw. Eng. Notes 17, 40–52.
Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X.S., Zang, B., 2013. Vetting undesirable behaviors in android apps with permission use analysis. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security. ACM, Berlin, Germany, pp. 611–622. doi:10.1145/2508859.2516689.

Hamid Bagheri is an Assistant Professor in the Computer Science and Engineering Department at University of Nebraska–Lincoln. Previously, he was a postdoctoral researcher in the School of Information and Computer Sciences at University of California, Irvine. He has also visited Massachusetts Institute of Technology as a postdoctoral research fellow. He received his Ph.D. in Computer Science from University of Virginia, a M.Sc. in Software Engineering from Sharif University of Technology, and his B.Sc. in Computer Engineering from University of Tehran. Hamid is broadly interested in software engineering, and particularly in practical software analysis and synthesis using concepts from fields like formal methods, program analysis, model-driven development, and software architecture. He has been a finalist at the ACM student research competition. His publications in several conferences have been recognized as best papers. Hamid has served on the program committee of several major conferences, and reviewed for multiple journals. He is a member of ACM and IEEE.

Joshua Garcia received three degrees from the University of Southern California: a B.S. in computer engineering and computer science, an M.S. in computer science, and a Ph.D. in computer science. He is an assistant project scientist at the Institute for Software Research, which is located at the University of California, Irvine. He conducts research in software engineering with a focus on software security, software testing, and software architecture.

Alireza Sadeghi is a Ph.D. candidate in the Department of Informatics of ICS School at the University of California, Irvine. His research interests focus on software engineering, specifically, application of program analysis in security and energy consumption assessment of mobile applications. Sadeghi received the B.Sc. degree in computer (software) engineering and M.Sc. degree in information technology from Sharif University of Technology in 2008 and 2010, respectively. He is a member of ACM and ACM SIGSOFT.

Sam Malek is an associate professor in the School of Information and Computer Sciences at the University of California, Irvine. He is also the director of Software Engineering and Analysis Laboratory and a faculty member of the Institute for Software Research at UCI. Previously he was an Associate Professor in the Computer Science Department at George Mason University. He received the B.S. degree in Information and Computer Science from the University of California, Irvine, and the MS and Ph.D. degrees in Computer Science from the University of Southern California. His general research interests are in the field of software engineering, and to date his focus has spanned the areas of software architecture, autonomic computing, software security, and software analysis and testing. He has received numerous awards for his research contributions, including the U.S. National Science Foundation CAREER award, GMU Emerging Researcher/Scholar/Creator award, and the GMU Computer Science Department Outstanding Faculty Research award. He is a member of the ACM, ACM SIGSOFT, and IEEE.

Nenad Medvidović is a professor in the Computer Science Department at the University of Southern California. Medvidović is the founding Director of the SoftArch Laboratory at USC. Previously he served as Director of USC's Center for Systems and Software Engineering. Medvidović is currently serving as Chair of ACM's Special Interest Group for Software Engineering (SIGSOFT) and was until recently chair of the Steering Committe for the International Conference on Software Engineering (ICSE). He was the program chair of several conferences, including ICSE 2011, and has served as associate editor of several journals. Medvidović is a recipient of the U.S. National Science Foundation CAREER award, the Okawa Foundation Research Grant, the IBM Real-Time Innovation Award, and the USC Mellon Mentoring Award. He is a co-author of the ICSE 1998 paper that was recognized as that conference's Most Influential Paper. He is a co-author of a textbook on software architectures. Medvidovi is an ACM Distinguished Scientist and an IEEE Fellow.