2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

Inter-app Communication in Android: Developer Challenges

Waqar Ahmad, Christian Kästner, Joshua Sunshine, and Jonathan Aldrich
Carnegie Mellon University, USA

ABSTRACT

The Android platform is designed to support mutually untrusted third-party apps, which run as isolated processes but may interact via platform-controlled mechanisms, called *Intents*. Interactions among third-party apps are intended and can contribute to a rich user experience, for example, the ability to share pictures from one app with another. The Android platform presents an interesting point in a design space of module systems that is biased toward isolation, extensibility, and untrusted contributions. The Intent mechanism essentially provides message channels among modules, in which the set of message types is extensible. However, the module system has design limitations including the lack of consistent mechanisms to document message types, very limited checking that a message conforms to its specifications, the inability to explicitly declare dependencies on other modules, and the lack of checks for backward compatibility as message types evolve over time. In order to understand the degree to which these design limitations result in real issues, we studied a broad corpus of apps and cross-validated our results against app documentation and Android support forums. Our findings suggest that design limitations do indeed cause development problems. Based on our results, we outline further research questions and propose possible mitigation strategies.

1. INTRODUCTION

The Android platform is designed for mutually untrusted third-party contributions ('apps') based on the idea of mostly isolating them from each other while allowing only restricted forms of platform-controlled inter-process communication. At the same time, interactions among third-party contributions are intended and contribute to a rich user experience, for example, the ability to invoke a map with a specific contribution from an app, or to share pictures from one app with another. In contrast to platforms like WordPress or Eclipse, third-party contributions have significantly fewer opportunities to interact with each other, which prevents many malicious interactions, but also significantly restricts tight integration and intended interactions, thus potentially limiting developers and throttling innovation. The Android platform presents an interesting design in a larger design space of module systems that is biased toward isolation, extensibility, and untrusted contributions. In this paper, we investigate how developers interact with Android's module system and its inter-process-communication mechanisms, studying their limitations to better understand the implications of the design and opportunities for improvement regarding app interaction, reuse, and evolution.

Android apps are modules that are executed in isolation from each other in separate processes. The Android platform provides a mechanism to exchange messages among modules through an inter-process communication API with messages coined *Intents*. The intent mechanism essentially provides message channels among modules, in which the set of message types is extensible. The current design of the Android module system has the following potentially problematic decisions:

- P1 To interact, multiple modules (potentially unknown to each other and mutually distrusting) need to agree on message types. While some message types are documented by the platform, there is no consistent mechanism of documenting, publishing, or negotiating message types.
- P2 Conformance to message type specifications is not checked by the platform but left to individual applications; message parameters are sent by untyped key-value maps. If an application sends a message without the expected or with incorrect parameters, the receiving module needs to resolve the problem. Combined with a lack of consistent documentation, it can be difficult to identify the expected message parameters for a sender.
- P3 Modules cannot declare dependencies to other modules. Modules need to prepare for the possibility that the messages they send are not received. In different installations, different modules may receive a given message (and may have different expectations toward that message type).
- P4 Modules and thus their expected message types may evolve independently from each other. Combined with the lack of versioning and conformance checking, interactions may stop working with the update of a module in a system.

In support forums such as Stackoverflow.com, we observe many questions that relate to these design decisions. For example, developers seek information about which parameters to pass, or why certain messages stopped working after an

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MSR'16, May 14-15, 2016, Austin, TX, USA

© 2016 ACM. ISBN 978-1-4503-4186-8/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2901739.2901762

update. However, before suggesting concrete interventions or alternatives to Android's module system, our goal is to better understand how developers use the module system and to which degree the design decisions mentioned above surface in issues. We statically analyze a corpus of 52535 Android apps regarding how they use inter-process communication and which message types they use and cross-validate our results with other sources, such as documentation and posts in support forums. We additionally studied the evolution of 6171 apps for which we have at least three revisions in our corpus. Specifically, we ask the following five research questions regarding inter-app interactions, community processes to agree on message types, and app evolution:

- Inter-app interactions
  RQ1: How are message types specified in practice?
  RQ2: How do developers use undocumented message types?
- Community processes to agree on message types
  RQ3: Are there message types that are received by different apps?
  RQ4: How do popular received message types emerge?
- App evolution
  RQ5: Do apps frequently adopt new message types or drop previously supported message types?

Our results indicate that developers want their apps to interact with other apps but struggle with finding information about third-party-contributed messages types. We find that many third party developers do not document message types and those who document, do in ad-hoc ways. Developers fall back to various reverse-engineering approaches to extract information about undocumented message types. Moreover, we have also discovered that even though platform-defined message types are commonly used, there are also a significant number of third-party-contributed message types that are adopted by multiple apps; such message types typically involve dominant, popular apps and documentation. Furthermore, adoption and removal of both received and sent message types is a common phenomenon as apps evolve over time. Combined with the lack of versioning and conformance checking, removals of received message types pose serious challenges for inter-app interactions and additions may remain underused or misused. Our results indicate several implications of Android's design decisions, and we suggest paths toward mitigating them with future research.

While our study is about the specifics of the Android platform, it can be considered as a case study in the larger context of module systems for software ecosystems. Composing and building upon more or less trusted third-party contributions is a common theme across many software ecosystems, including Eclipse, WordPress, node.js, R/CRAN, and many others. Lessons learned from Android's design will be useful for making deliberate design decisions for module systems of other software ecosystems.

In summary, we make the following contributions:

- We identify problems regarding inter-app interactions, grounded in Android's design decisions and evidence from developer forums.
- We answer 5 research questions regarding *inter-app interactions*, *community processes to agree on message types*, and *app evolution* with data extracted from a corpus of 52535 Android apps and from documentation and support forums.
- We propose mitigation strategies to address identified challenges without invasive changes to the Android's module system.
- We release a database (https://archive.org/details/interapp) with all extracted facts from our corpus, allowing others to study further aspects of inter-app communication and app evolution.

2. APP COMMUNICATION CHALLENGES

Composing and building upon more or less trusted third-party contributions is a common theme across many software ecosystems, including Eclipse, WordPress, node.js, R/CRAN, and many others. *A software ecosystem is the interaction of a set of actors on top of a common technological platform that results in a number of software solutions or services* [24] . At the heart of software ecosystems are technological platforms that enable a variety of players including businesses, developers, and users to interact with each other and develop innovative solutions. A key component of these platforms is a module system that allows software modules developed by diverse developers to interact with each other. For instance, in Android, modules interact with core framework of the module system but also with each other through an inter-app communication mechanism.

Android is designed to encourage inter-app interactions in an extensible and reusable way. The documentation describes the design rationale as follows: *Put together, the set of actions, data types, categories, and extra data defines a language for the system allowing for the expression of phrases such as 'call john smith's cell'. As applications are added to the system, they can extend this language by adding new actions, types, and categories, or they can modify the behavior of existing phrases by supplying their own activities that handle them.*[1]

The goals of our study are to understand challenges that developers face when working with inter-app communication, to identify causes of those challenges, and to propose mitigation strategies. We derived the research questions in several iterative cycles in which we searched developer forums for issues that are frequently discussed and relate them to design decisions in the Android platform (particularly the design decisions that differ from other module systems; *P1–4* listed in the introduction). This way, we identified pain points and relevant research questions that we explore systematically in this paper. In Figure 1, we summarize how design decisions, developer challenges, and research questions relate.

2.1 App Communication Mechanisms

To provide a context for developer problems, we give a brief overview of the technical infrastructure that the Android platform provides for inter-process communication, using messages called *Intents*. Android distinguishes between *explicit intents* that are addressed toward specific components and *implicit intents* that are dispatched by the platform depending on the message's name and attributes; multiple apps might be able to receive the same message. Explicit intents are generally used for communications within an app, whereas implicit intents are generally used for inter-app communication; hence, we focus on the latter.

Implicit intent messages have a name (*action id*) and can have additional attributes (e.g., *data types*, *URIs*, and

[1]http://developer.android.com/reference/android/content/Intent.html

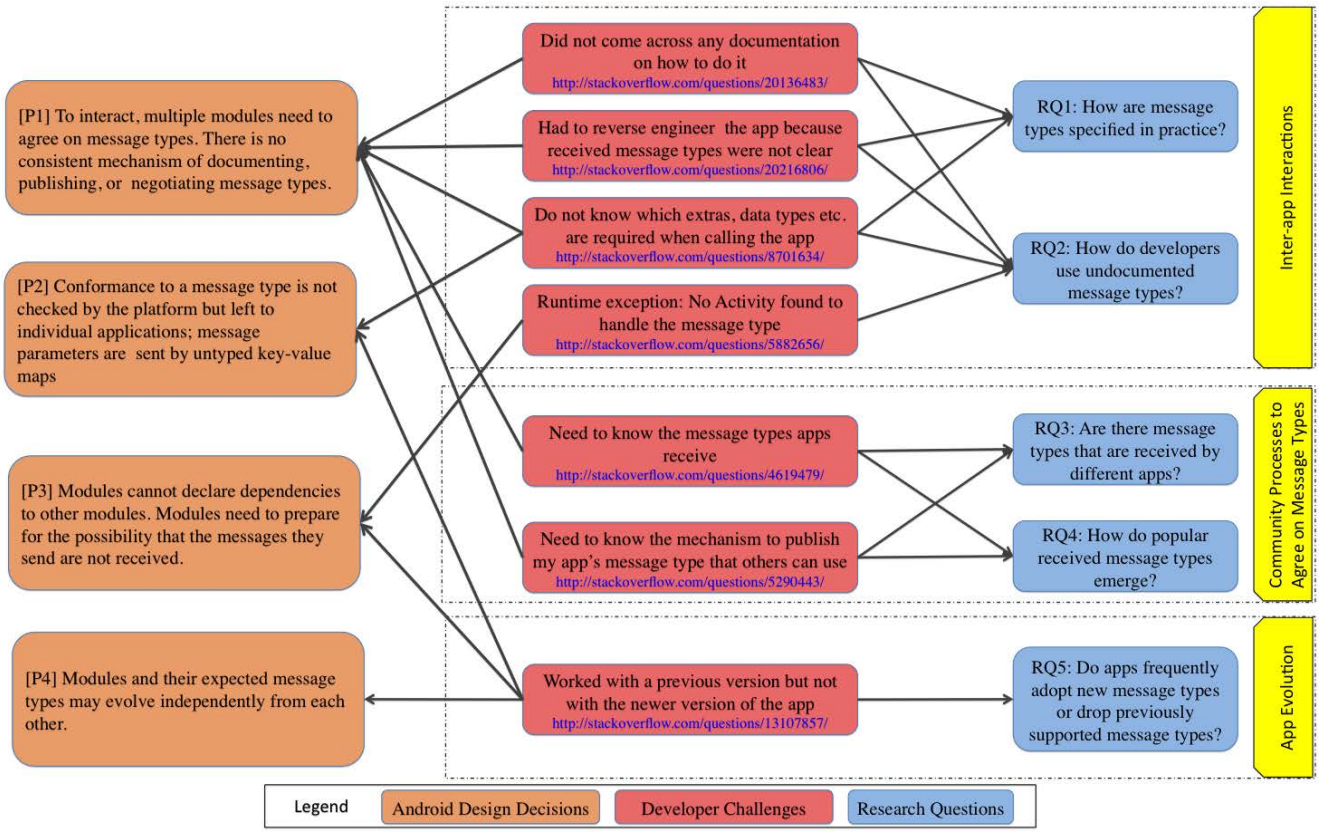


Figure 1: Design Decisions, Developer Challenges, and Research Questions: An Overview of our Study. Each of the http links represents just one case for the related developer challenge.

extras, the latter is a set of key-value pairs). For example, a message with the name `android.intent.action.EDIT` may additionally describe its data type as an image and provide an URI to access the image to be edited as well as the name of the app sending it. The set of message types is unlimited and messages are not formally specified. The Android documentation lists a number of common message names, and many of those are used frequently in built-in applications, but developers can send and receive messages with different names as well, and even send and receive messages with the same name but different attributes. In this paper, we distinguish between *platform-defined message types* for message types documented in the official Android developer documentation and *third-party-contributed message types* for all others.

Mechanisms for sending and receiving messages are defined by the platform, thus enabling inter-operability, reusability, and evolution of apps. Apps send messages with a platform API, including methods such as *startActivity*, *startService*, and *sendBroadcast* [20]. Apps declare as part of their implementation which messages they are able to receive (called *intent filters*). This is typically defined in the app's manifest file which specifies the messages the app accepts based on the message types and message attributes. For each message, the platform identifies all apps able to receive it; if multiple apps can receive the message, the platform picks a receiver, typically by asking the user.

2.2 Observed Challenges

In our exploratory search for developer challenges regarding inter-app communication in developer support forums, we have discovered a number of common challenges that seem to be caused by platform design decisions and that motivate our research questions. In the following, we highlight a few representative cases and list the remaining ones with pointers to research questions and Android's design decisions in the middle column of Figure 1.

Developers frequently search for information about message types to communicate with other apps. A lack of documentation appears to be a common source of frustration, as expressed in this support request: "*Samsung's TWLauncher allows apps to create badge counts on app icons. This is completely undocumented ANYWHERE and only a handful of apps are using this (Facebook, eBay to name a couple). How do you use this functionality to add a count to your application icon?*"[2] Similarly, many developers experience runtime exceptions when providing incorrect or insufficient message attributes, because there is no platform mechanism to check correctness of messages. The differences between expected and actual attributes can be subtle. For example, "*EXTRA_MAIL should correspond to a String[], not just a String as shown here.*"[3] These developer challenges are related to Android's design decisions *P1, P2, and P3* (see

[2]http://stackoverflow.com/questions/20136483
[3]http://stackoverflow.com/questions/8701634

Figure 1). We want to investigate these challenges further and raise the following two research questions: *RQ1: How are message types specified in practice? RQ2: How do developers use undocumented message types?*

The ability of multiple apps to share message types is an important feature of Android's design that allows an app to substitute for the behavior of another app. Again, app developers struggle with identifying the message types received by existing apps to mirror them. At the same time, they seek ways to standardize and publish message types that their own apps might create, as shown in this post: "*Is there an Intent-database where one can search for applications that publish common services? For example I could have an idea about a filter that could be applied to photos in a photo-application, but under what intent should I publish my filter so that other applications can find it and use it?*"[4] This phenomenon is related to Android's design decision *P1* (see Figure 1). In order to investigate more closely the current practices and mechanisms to document, publish, and negotiate message types, we raise the following two research questions: *RQ3: Are there message types that are received by different apps? RQ4: How do popular received message types emerge?*

Finally, apps and their expectations toward message types may evolve independently from each other. Combined with the lack of versioning and conformance checking, interactions may stop working with the update of an app in the system. We found developers struggling to cope with changes in app message types on support forums: "*Till few days ago, in order to show to my users another user profile I used the following solution: [...] The last facebook app update [...] made this solution obsolete.*"[5] This phenomenon is related to Android's design decisions *P2, P3, P4* (see Figure 1). To investigate the severity of these issues, we assess evolution characteristics of sent and received message types across time and ask a final research question: *RQ5: Do apps frequently adopt new message types or drop previously supported message types over time?*

3. METHODOLOGY

To study inter-app communication, we statically analyzed which apps can receive and send which kind of message types on a large corpus of apps. We complement the data with qualitative analysis of documentation and support forums.

3.1 Corpus

To answer our research questions, we need a corpus of apps that (a) is large enough to have a chance of identifying common communication patterns and that (b) contains multiple revisions of apps to study their evolution. To that end, we have collected a large corpus of apps and have combined several strategies to collect historical revisions for a subset of these apps. We collected apps from the following sources:

- We seeded our corpus with 75,000 app revisions collected from the Play store by collaborators for prior studies [3, 34].
- We iteratively downloaded free apps from the Play store with an automated script. The script checks the availability of new revisions of apps in our corpus and downloads them. Subsequently, the script would randomly navigate the top apps in each category and download up to 30 apps that are not in our corpus. We executed the script daily from Jun. 2015 to Jan. 2016.
- We downloaded all apps from the open-source app store *F-Droid*.[6] In contrast to the Play store, old revisions are also available on F-Droid, which we collected as well. In total, we downloaded 8935 revisions of the 1740 apps in F-Droid.
- We extracted apps from seven factory images that Google provides for its own phones, ranging from images for Android 2.3.7 to 5.0.2. Extracting apps from older images provided access to historic revisions of standard apps on the Android platform, including open-source apps such as Contacts and Calendar and closed-source apps such as Chrome and YouTube.

In total, our corpus includes 52535 unique apps. For 6171 of those apps, we were able to collect 3 or more revisions; for 659 apps, we collected more than 7 revisions.

3.2 Analyzing Inter-App Communication

To analyze which apps can receive and send which kinds of messages, we used an existing static analysis tool *IC3* [30], which was designed initially to detect security-relevant information flow across apps. *IC3*, similar to its predecessor *Epicc* [31], statically analyzes the app's byte code and manifest files to identify which intent filters are declared (i.e., which message types an app can receive) and which intents (both explicit and implicit) are sent (i.e., which message types an app can send). IC3's analysis is inter-procedural, flow- and context-sensitive; it can identify sent messages including data types and extras for most cases. For received messages, it identifies only message types as declared in the app's manifest file, which does not necessarily include details about all attributes. We investigated this tool and found it effective for our purposes.

We analyzed 88353 app revisions of our corpus with *IC3* and stored the results in a database for further analysis. The analysis timed out (we terminated after 30 minutes) for 173 app revisions, leaving us with 52535 unique apps and a total of 88180 revisions for our study.

3.3 Other Sources

To supplement and cross-validate our results, we complemented data gathered from statically analyzing apps with a (mostly manual) analysis of support forums and online documentation. For instance, we searched for documentation of a sample of commonly used message types and investigated support forums for hints about how developers gather information about undocumented message types. We will describe the corresponding analysis and sampling steps below.

4. INTER-APP INTERACTIONS

As discussed above, module developers want to implement inter-app interactions, but repeatedly struggle with implementing inter-process communication correctly. Multiple apps need to agree on shared message types, both when calling and receiving messages, which includes not only a name ('action identifier)', but also various untyped and sometimes optional parameters ('data type', 'extras', etc.). As message types are not specified or checked by the platform and an app's expectations toward a message type may change over

[4]http://stackoverflow.com/questions/5290443
[5]http://stackoverflow.com/questions/13107857
[6]https://f-droid.org

time, developers often report difficulties in developing rich interactions with other apps. In this section, we explore two questions regarding community practice in documenting and discovering message-type details.

RQ1: How are message types specified in practice?

Android does not provide any formal mechanism or tooling for documenting message types, including their parameters, expected by an app or even multiple apps. In contrast to many other ecosystems, a structured or tool-supported mechanism for specifying message interfaces (e.g., comparable to *JavaDoc* for documenting Java interfaces, to Eclipse's extension point schema, or to *WSDL* for specifying web services) does not exist in Android. Instead, we investigate the documentation of a large number of commonly used message types.

We analyzed the use of messages in our corpus of Android apps to identify the names of message types ('action identifier') that are used by most apps in our corpus. We first collected the 300 message-type names that the most apps in our corpus can receive (each receivable by at least 10 or more apps in our corpus) and the 300 message-type names that the most apps in our corpus send (each sent by more than 6 apps in our corpus), resulting in a total of 522 distinct message-type names.

In an automated process, with a simple text search, we identified all message types that are documented as part of the official Android documentation (at http://developer.android.com and https://developers.google.com/android/): 203 out of the 522 considered message types are *platform-defined*, whereas we classify all others as *contributed* message types. From the contributed message types, we randomly sampled 100 message types for manual analysis. For each message type, we searched the message type's name using standard web search engines. We used a wide definition of documentation, including more formal documentation as well as blog posts and publicly available example programs.

All message types automatically classified as *platform-defined* are described in the platform documentation. Among these, 41 commonly used platform-defined message types are centrally documented in particular detail, specifying all parameters including message-type name, data types, and extras, whereas all others are documented as part of the corresponding API; for example, Bluetooth-related message types are documented with the Bluetooth API documentation.[7]

In our sample of popular *contributed* message types, we found documentation for 45 of 100 message types. For 30 message types, the app's developer published documentation on their personal website; for 5 message types, a third party documented the message type, for example on a blog; and for 10 message types, the source code of an app that implements the message type was available as a reference. We observed that developers used a variety of ways to publish message-type documentation. In 13 of the 45 instances, documentation followed a pattern similar to JavaDoc, describing message names and corresponding parameters; in 14 cases, message names and parameters were described in a tutorial with sample code; in 8 cases, sample code about how to send or receive a message was published; whereas in the remaining 10 cases, the source code of apps using the message was the only source of documentation.

Type of documentation	Number of cases in our sample
Message type and extras with types and descriptions	13
Message type and sample code	8
Tutorial and sample code	14
App source code only	10
Undocumented	55

Table 1: Documentation of third-party-contributed message types

Looking for a relationship between documentation and popularity of contributed message types in our sample, we found that documented messages types were received by more apps (36 apps on average receiving the sampled documented contributed message types versus 19 apps for undocumented contributed message types; statistically significant with $p < 0.01$ according to a t-test), but not necessarily called by more apps (51 vs 77 apps on average, but without clear trends, $p > 0.5$). We found both documented message types implemented by only 10 apps and undocumented message types implemented by over 100 apps. We conjecture that documentation contributes to the popularity of message types but is likely not the driving factor.

RQ2: How do developers use undocumented message types?

Given that a large number of contributed message types were not visibly documented despite some popularity with developers, we further searched for clues about how developers cope with such a lack of specification. In addition to the discussions uncovered when framing the problem in Section 2.2, we further searched in support forums for the 55 undocumented message types and identified six corresponding discussions on *Stackoverflow* and *XDA*.

We identified the following strategies:

- *Extracting manifest files* and *decompiling* the source are common strategies to extract additional information about a message type from an app's implementation (which we also used to gather data about our corpus). The manifest file will provide the names of message types an application can receive, whereas the code can provide more details about which message parameters are accessed or how messages are sent within an app. For example, the apktool for extracting manifest files was mentioned in a question about message types supported by the Facebook app.[8]
- *Other apps* that implement the same message type can serve as reference for identifying message-type information, thus developers build on the reverse engineering of the work performed by others. For example, developers have analyzed how Sony's email app provides badges to Sony's launcher, which was otherwise undocumented.[9] Developer discussions on support forums should not, however, be confused with the documentation of the message types by third party blogs mentioned earlier. While third party blogs provide information in a relatively structured way, support forums generally provide workarounds for problems in a specific context that may

[7]http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html

[8]http://stackoverflow.com/questions/4445944

[9]http://stackoverflow.com/questions/20216806

work for some developers but not necessarily for others without pointing out the underlying reasons.
- Registering a *dummy broadcast receiver* is a common way to intercept broadcast messages with a known name to dynamically observe which message parameters ('extras') are passed between apps.[10]

Reverse engineered information about message types is sometimes posted as part of the discussion, but not documented systematically. In fact, developers have to repeat the reverse engineering task if an app evolves in an incompatible way, as for example discussed for the Facebook app.[11]

Our analysis provides evidence for problems caused by the lack of message-type specification and the desire to interact with other apps nonetheless. Our sample is too small to judge how common each strategy is. However, several of these strategies are well known and applicable for many apps, so it is not surprising that many undocumented message types are nonetheless implemented by many apps, even without public discussion.

Our results indicate that a repository with message type specifications could reduce a pain point of Android developers that aim for interactions. The closest available repository for contributed message types right now is OpenIntents,[12] but it specifies only a small number of message types and does not collect reverse engineered message types. Moreover, it does not track changes as message types evolve over time. For each message type, the documentation provides a brief description and sample code for sending and receiving it, and the names and types of the parameters. A static analysis tool could provide an initial data set for such a repository.

5. COMMUNITY PROCESSES TO AGREE ON MESSAGE TYPES

A powerful mechanism of Android's module system is that the target of a message may be determined at runtime and multiple modules may be able to respond to the same message (called an 'implicit intent' in the Android terminology). While two modules can always agree on specific message types to be exchanged between them, community agreement on specific message types allows easy system extensibility, where a module provider may react to the same message that other modules can also receive, thus allowing a developer to provide alternatives to existing modules. Similarly, this scheme enables seamless integration into existing extension points, for example through a generic sharing mechanisms over a shared message type, instead of providing a separate sharing button and messages for every potential module some data could be shared with. Note, the concept that many apps can receive the same message from an extensible set of community-defined message types is common in event-based systems, such as ROS [32], but not broadly used for interactions in software ecosystems like Eclipse or WordPress.

Technically, we look at only the name of the received message types, called 'action identifier', to simplify our analysis. We analyze activities, services, and broadcast receivers, but not content providers since they require unique identifiers. We count each message type at most once per app even if it implements it multiple times. In a manual post-processing step, we filtered message types that are implemented by multiple apps from the same provider (as apparent from the app's name or app-store metadata).

To benefit from this flexibility, many developers have to agree on receiving messages that use the same shared message type. As there is no formal mechanism to support this process, we explore further the message types that are shared by different apps and how new shared message types emerge with the next two research questions.

RQ3: Are there message types that are received by different apps?

To address this question, we first identified which apps in our corpus declare to receive which message types. This will indicate whether multiple module developers, in practice, have agreed on common message types shared across many modules.

The 52535 apps in our corpus declared that they could receive a total of 41180 distinct message types. Over 2965 (7 percent) of these message types can be received by multiple apps in our corpus. However the distribution of how many apps can receive each message type are heavily skewed, with the most popular message type '`android.intent.action.SCREEN_OFF`' receivable by 9836 apps, whereas the 30th most popular '`android.intent.action.MEDIA_UNMOUNTED`' is received by only 466 apps and 98.7 percent of all message types are not received by more than 10 apps. In Figure 2, we show this distribution for the 300 most popular message types.

Investigating the data it is obvious and unsurprising that the most popular message types are all messages types documented in the platform documentation and implemented by many apps that come with a default installation of Android. To identify whether the community would agree on message types not defined by the platform, we again distinguish message types into platform-defined and third-party contributed, considering a message type as platform-defined if its name is mentioned in the official Android documentation. We found that although almost half of the top 300 message types are third-party contributed, the top 59 message types, including all message types that are implemented by over 500 apps, are all platform defined. The most frequently received *contributed* message types are '`com.google.zxing.client.android.SCAN`' and '`com.amazon.inapp.purchasing.NOTIFY`' with 165 and 115 apps implementing them respectively. In Figure 2, we highlight all contributed message types among the 300 most popular message types.

Our data indicates not only that platform-defined message types are used across a significant percentage of Android apps, but also that a significant number of contributed message types have achieved some popularity in our corpus. This indicates that contributed message types emerge for common problems in the absence of corresponding platform-defined message types (e.g., for barcode scanning). This leads us to our next question about how such message types emerge without platform support for coordination.

RQ4: How do popular received message types emerge?

Finding that none of the most received message types, but still many message types received by 10 to 100 apps in our corpus, are contributed by the community, we decided to further investigate how those message types emerged and techniques and practices that developer may use to establish communication among many independent apps.

[10] http://stackoverflow.com/questions/10510292
[11] http://stackoverflow.com/questions/13107857
[12] http://www.openintents.org/intentsregistry

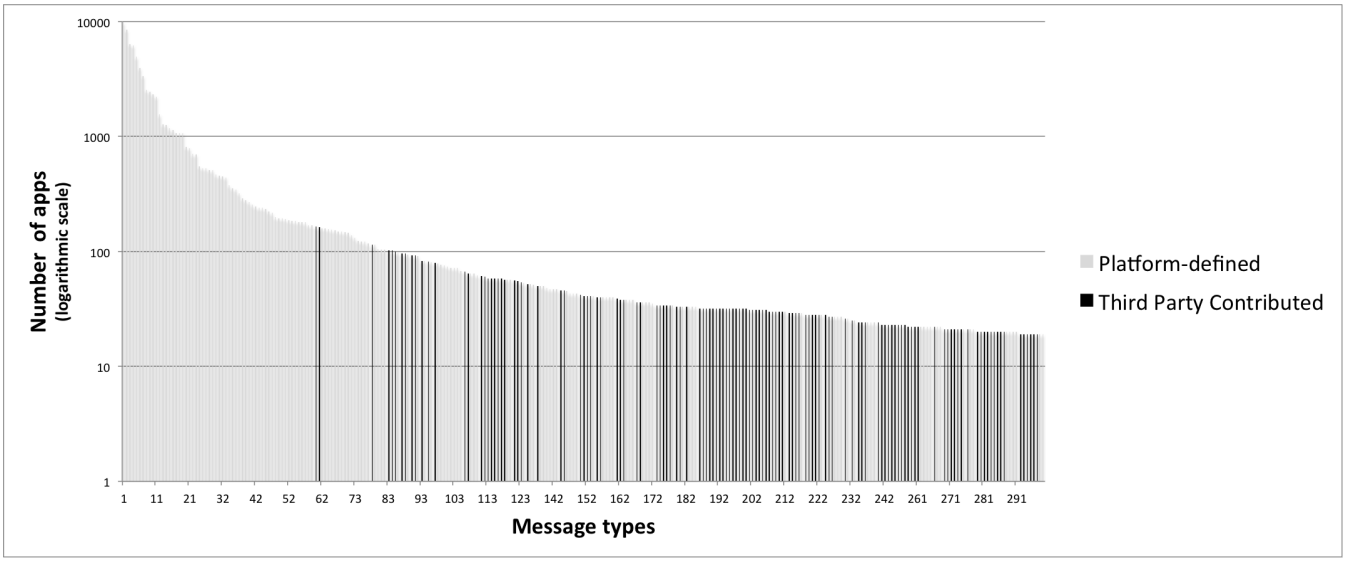


Figure 2: The distribution of the message types by the number of apps that receive them.

We proceeded as follows: We randomly sampled 20 contributed messages types listed in Table 2, among the 300 most received message types (see RQ1 and RQ3). In our sample of top 300 message types, we had third-party-contributed message types received by a wide range of apps from 10 to 165. In order to make our sample representative, we divided our sample in subgroups with each subgroup covering a range of 10, e.g., 11-20, 21-30 and so on. We selected top two message types from each subgroup. We searched for the names of those message types online (support forums, Github, app stores) and investigated the apps receiving them. Where available, we studied online documentation and source code related to each message type. We also collected the number and kinds (activity, service, broadcast receivers) of callers and callees for each of the message types.

We iteratively discussed our findings and grouped them into various patterns. In our sample, we identified three patterns for how contributed message types emerge that explain most of the sampled message types:

- For 9 of the 20 sampled message types, the app(s) sending the message seem to dominate the relationship and are a likely cause for the popularity. These can be considered as apps providing extension points through messages, whereas other apps receiving those messages act as plugins. One example in this category is the message type `org.adw.launcher.THEMES`, sent by a popular Android app *ADW.Launcher* (with over 10 million downloads), allowing users to customize their devices with a variety of themes and wallpapers. While this app provides its own themes and wallpapers, it can access themes and wallpapers provided by other apps receiving its own message type, documented in a guide online.[13] Many developers have implemented the message type to provide additional themes. Eight other message types in our sample follow the same pattern. In each case, the popularity of receiving a message type is driven by a single very popular app, which explicitly documents extension points.
- For 7 of the 20 sampled message types, a common library or API used by multiple apps likely triggered the common message type. For example, in many installations, Android does not provide a default mechanism to scan barcodes; in order to not ask users to first install a barcode scanning app, many apps include a library *ZXing* to provide their own barcode-scanning functionality; following the library's documentation many apps publish this barcode-scanning functionality to other apps as well. Six other message types in our sample follow the library-driven pattern.
- For 2 message types, multiple apps provide alternatives to a common, documented, and frequently used message type. In this case, the message type `org.openintents.action.PICK_FILE` supports opening a dialog to select a file from the file system. This message type is received by a popular app *OI File Manager* (over 5 million downloads) and documented publicly (on a platform called *OpenIntents*), but also implemented by several alternative file-manager and other apps (e.g., *File Commander*, and *Total Commander*). In our sample, we found only two message types that followed the same pattern. Therefore, we cannot make a broad conclusion from only two cases but we conjecture that the following four factors are important for message types emerging this way: (a) similar functionality is not yet provided by platform-defined message types, (b) an app popularizes the message type, (c) many or popular apps call this message type, and (d) the message type is documented explicitly or source code of a reference implementation is available.

For 2 of the message types in our sample, we could not judge how they have emerged. Since message types may become popular in a variety of ways, app developers should carefully define message type specifications because it may be quite problematic to update them at a later stage when a message type has become popular.

Apart from the *OpenIntents* efforts, we did not see any deliberate effort to coordinate the definition of message types.

[13]http://adwthings.com/launcher/adw-theming-guide

Message Type	Receivers	Callers	Pattern
com.google.zxing.client.android.SCAN	165	147	Library
com.amazon.inapp.purchasing.NOTIFY	115	0	Library
com.startapp.android.CloseAdActivity	102	2	Caller
com.appenda.INSTALL_INTENT	101	91	Caller
com.appenda.AppNotify	96	0	Unknown
com.gau.go.launcherex.theme	93	0	Caller
com.google.zxing.client.android.ENCODE	83	99	Library
org.adw.launcher.THEMES	67	0	Caller
com.adamrocker.android.simeji.ACTION_INTERCEPT	61	6	Caller
com.xtify.android.sdk.SEND_SETTINGS	58	52	Library
com.inmobi.share.id	57	35	Library
com.urbanairship.airmail.END_REGISTER	46	0	Library
com.gt.slinglabs.SlingNotify	41	0	Unknown
org.OpenUDID.GETUDID	39	0	Library
com.sonyericsson.extras.liveware.aef.registration.ACCESSORY_CONNECTION	33	0	Caller
vpn.connectivity	29	4	Caller
com.htc.music.playbackcomplete	29	0	Caller
org.openintents.action.PICK_FILE	19	37	Callee
jp.r246.twicca.ACTION_SHOW_TWEET	16	0	Caller
org.openintents.action.PICK_DIRECTORY	10	22	Calllee

Table 2: Contributed message types, number of apps in our corpus that receive and send them, and the patterns that helped them become popular.

Nonetheless, we could identify several patterns in how multiple apps adopt receiving the same message types, typically involving dominant, popular apps and documentation. In this context it is not surprising that contributed message types are much less adopted than platform-defined ones. Our findings raise interesting challenges about how to support the community in defining and reusing message types.

A repository for intents has interesting parallels to UDDI, a standard for description and runtime discovery of web services [27].While UDDI was embraced initially, it has received little adoption outside individual companies. We think that one reason that UDDI was not widely adopted was that it tried to achieve standardization in highly dynamic business environments. We argue that documentation and awareness of change, while still allowing a community process and extensibility in the messaging mechanism is more promising than enforcing standardization and constraining change.

6. APP EVOLUTION

Even when multiple apps agree on shared message types, they can evolve independently from each other. In fact, the Android platform has no mechanism to express versioned dependencies on other apps and no mechanism to enforce any consistency of attributes in message types. As we have seen in Sec. 2.2, we have seen some frustration from developers that interactions stop working with updates within the ecosystem. With our final research question, we attempt to explore the problem by characterizing how frequently apps change the message types they use for interactions.

RQ5: Do apps frequently adopt new message types or drop previously supported message types over time?

We investigate both how frequently apps evolve to *receive* new message types and how often they evolve to *send* new message types. For sent message types, we additionally explore how frequently attributes of those messages are changed, i.e., developers send additional (or fewer) extra attributes with a message in a new revisions.

We quantified change on all 6171 apps in our corpus for which we have at least three revisions (see Sec. 3.1). For each of these apps, we can build two tables that show which message types are potentially received and sent in which revision, as exemplified in Table 3 for the open-source app *Open Explorer*. In this example, some message types, such as `org.openintents.action.PICK_FILE`, were only adopted in later revisions, whereas others were removed, and yet others were supported across all versions. We distinguish message types that were added (switching from unsupported to supported at some point in their evolution), removed (switching from supported to unsupported at some point), consistent (supported in all revisions), and other (switching between supported and unsupported multiple times).

On the receiving side, we found that 38 percent of the apps added at least one new received message type and 18 percent removed at least one during the evolution we observed. On average an app adds 0.3 and removes 0.1 received message types per revision. The removal, in particular, can be concerning to apps that may have sent messages with this type previously. We also found that about 9 percent of the apps added a message and then removed it in a later version, or removed a message type and then added it again later.

On the sending side, we observed a similar picture. We found that 36 percent of the apps added at least one new sent message type and 16 percent removed at least one during the evolution we observed. On average an app adds 0.17 and removes 0.08 sent message types per revision. These numbers show that apps change frequently to adopt new communication patterns and change existing ones, emphasizing the trend toward rich interactions. In addition, we speculate that the similar rates of change indicate a co-evolution of sent and received message types, but further study is needed to validate this hypothesis.

Furthermore, we also observed significant rates of change in attributes of sent messages (called 'extras' in Android). We found that about 14 percent of message types used across all apps were adjusted by the sending side during their evolution to include additional attributes, and attributes were removed from 7 percent of message types. This emphasizes again

Message-type name	V. 116 Oct-11	V. 125 Dec-11	V. 189 Sep-12	V. 194 Nov-12	V. 208 Jan-13	V. 212 Feb-13	V. 221 Mar-13
android.bluetooth.device.action.FOUND	•	•					
android.intent.action.MEDIA_MOUNTED	•	•					
android.intent.action.VIEW	•	•	•	•	•	•	•
android.hardware.usb.action.USB_DEVICE_ATTACHED		•	•	•	•	•	•
android.intent.action.MEDIA_SCANNER_STARTED		•					
android.intent.action.SEARCH		•	•	•	•	•	•
android.intent.action.EDIT			•	•	•	•	•
android.intent.action.PICK			•	•	•	•	•
org.openintents.action.PICK_FILE			•	•	•	•	•
android.nfc.action.NDEF_DISCOVERED						•	•
org.brandroid.openmanager.server_type							•

Table 3: A snapshot of the changes in received message types across different revisions of the app *Open Explorer*; • indicates revisions of the app that can receive a given message type

the importance and relative frequency of changes within a message type, about which we saw several messages in support forums (see Sec. 2.2).

Overall, our analysis indicates that adoption and removal of both received and sent message types is a common phenomenon, as is change of attributes within a message type. While additions in received message types are new contributions that open up new opportunities for inter-app interactions, their removals may limit the previously working interactions. Combined with the lack of versioning and conformance checking, removals pose serious challenges for inter-app interactions. Moreover, due to the lack of consistent documentation mechanisms, new contributions may remain underused or misused.

7. DISCUSSION

Android's module system has design limitations including the lack of consistent mechanisms to document message types, very limited checking that a message conforms to its specification, the inability to explicitly declare dependencies on other modules, and the lack of checks for backward compatibility as message types evolve over time. Our results indicate that developers want their apps to interact with other apps but struggle with finding information about third-party-contributed messages types. We find that many third party developers do not document message types and those who document do in ad-hoc ways. Developers fall back to various reverse-engineering approaches to extract information about undocumented message types. Moreover, we have also discovered that even though platform-defined message types are commonly used, there are also a significant number of third-party-contributed message types that are adopted by multiple apps; such message types typically involve dominant, popular apps and documentation. Furthermore, adoption and removal of both received and sent message types is a common phenomenon as apps evolve over time. Combined with the lack of versioning and conformance checking, removals of received message types pose serious challenges for inter-app interactions and additions may remain underused or misused. Our results indicate several implications of Android's design decisions, and we suggest future research directions to mitigate them.

7.1 Mitigating Challenges (Future Directions)

We conjecture that most of the discussed challenges can be mitigated with external tools, without invasive changes to the Android platform. We plan to build a set of three complementary tools to support developers with Android inter-app communication.

First, we plan to develop a standard format for documenting message types and a searchable online repository for message types, where developers can publish specifications of new message types as well as look for existing message types. The repository will also indicate which apps receive and send a given message type and can point out changes over time and variants and inconsistencies in how message attributes are used in practice. This repository can also be a central place to share reverse engineered information about undocumented message types.

Second, to populate the repository, we will build an extractor tool that will identify sent and received message types including their attributes. We will build on top of the *IC3* infrastructure for static analysis, but extract additional information, including which apps parse which extras on received messages. In addition, we will consider dynamic monitoring to gather samples. We will seed the repository with data extracted from our corpus and will update it with newer revisions of apps as they become available.

Third, we will create an IDE plugin for Android Studio to auto-complete and type-check message instantiations. The plugin will simplify creating correct messages by creating templates for select message types, that will include stubs for all relevant message attributes. In addition, it will point out when commonly used or recently introduced message attributes are missing or of the wrong type. It can encourage change awareness in an ecosystem that is changing in a decentralized fashion.

We hope that such tool support will encourage more inter-app interactions and reuse (even and especially with contributed message types), improve developer productivity, and foster community for inter-app interactions.

7.2 Threats to Validity

As in all studies, there are several potential threats to the validity of our results. First, the findings related to two of our research questions (RQ2 and RQ4) are derived from a fairly small sets of apps. These apps were sampled carefully, but they are nonetheless a tiny fraction of apps. Second, only one of the authors studied the online documentation and support forums in order to extract qualitative information; however, all of the authors iteratively discussed the results together. Third, the vast majority of apps in the dataset are free, and the dataset is therefore not a perfect sample of all Android apps. In addition, apps seeded from Berger's data set [3] are 5 years old, which was intentional to get

old revisions. Because of time constraints, apps from Berger were only analyzed if we had a newer version in some other part of our dataset or it happened to be early in the analysis queue. However, we are confident that the most important and widely used apps are included. Finally, *IC3* failed to successfully analyze a few apps (e.g., because they were obfuscated). We inherit problems from IC3 and they mostly represent the standard challenges of program analysis. The generalizability of our results to the full body of Android apps can only be interpreted in the context of these limitations.

The validity is also threatened by the use of intent action names as a proxy for message types (throughout the study aside from part of RQ5). Considering full message types between apps is much more complicated—intents include data and key-value pair parameters (extras), and non-intent communication via content providers is common. Studying the full richness of Android message types would result in a more detailed picture of inter-app communication and may affect some of our results. We believe that further studying message attributes and content providers in detail is a valuable future research direction.

8. RELATED WORK

We investigate the inter-app communication in the Android platform as a case study in the larger context of module systems for software ecosystems. Beyond technical aspects software ecosystems cover interaction of actors on top of a common technological platform and how they interact with that platform [15, 24]. In this context, the module system can play a central role in designing variability mechanisms and facilities for interactions for developers [3, 5].

A central challenge is intended and unintended interactions among modules. Under the label *feature interactions*, interactions among independently designed modules have been studied in depth [6, 29]. The key issue is that unexpected behavior may emerge when two separately tested modules are combined. Several studies have shown that integration and interaction testing in ecosystems is often neglected, resulting in frequent interaction faults [2, 10, 11, 28].

Module systems can play a powerful row in restricting and controlling possible interactions among modules. Research in module systems and modularity mechanisms has a long tradition and many different points in the design space have been explored, e.g., [1, 3, 4, 13, 19, 33]. Most work on traditional module systems requires a certain amount of central planning; in contrast, in this work, we are particularly interested in how developers define and adopt message types in a distributed environment without central control. In this context, module systems should help cultivating a culture of innovation in which a variety of collaborating or competing actors build diverse modules and define inter-module communication message types in a decentralized manner.

Specification of message types is a key element for developers to learn about the message types implemented by the modules developed by other developers. Many formalisms have been developed to more or less formally describe the format of messages among modules, including IPC [13], WSDL, and OMG CORBA IDL.[14] Android does not adopt a formal specification mechanism; instead, we investigated how developers document and discover message types in practice.

The Android ecosystem has attracted significant attention for empirical studies, including topics as varied as API evolution [21, 22, 25] and energy costs [12, 36]. More specifically, we are interested in inter-module communication, which has received significant attention from security perspective, because it can be used for covert communication, potentially circumventing Android's permission system. To that end, various static and dynamic analysis tools have been developed to extract information about inter-module communication and analyze information flow properties both statically [16, 20, 30, 31, 35] and dynamically [9, 23]. Such static analysis has also been used for fuzz testing intent implementations [23]. In this study, we reuse the existing static analysis tool *IC3* [30] to extract information about received and sent Intents, but we analyze the extracted data for different research questions.

The Android platform and Android apps evolve over time, as all other software systems. Software evolution has been studied in depth for many years [17, 26]. Changes in message types that apps receive and send are related to API evolution, which has been shown to be a significant problem and studied empirically [8, 14, 21, 25]. Software evolution in component based systems is also a challenging process [7, 18]. Previous work in this area cover the module systems, in which inter-module dependencies are explicitly defined [7]. We study the evolution of inter-module communication in Android in which third party contributions are common and module interactions are resolved at runtime without explicit specifications of inter-module dependencies.

9. CONCLUSION

Android supports inter-app communication among mutually untrusted third-party apps through a platform-defined message mechanism. Android's design encourages to extend the platform with new message types, but message types are only partially checked by the platform and often are not formally documented. To understand the implications of Android's design decisions for developers, we explored several research questions regarding the use, documentation, adoption, reverse-engineering, and evolution of message types on a large corpus of Android apps. We found that interactions are in fact common, but also that third-party-contributed message types are often undocumented and can cause frustration for developers attempting to use them. Apps can often receive the same message types, but adoption of contributed message types is hindered by the lack of a central publishing mechanism. Finally, we found strong evidence that apps frequently adopt new messages types and existing message types are sent in different ways. These results encouraged us to propose lightweight tool-based mitigation strategies consisting of a central message-type repository, integrated with an IDE, and seeded with automatically extracted information.

10. ACKNOWLEDGMENTS

Support is acknowledged from the U.S. Department of Defense through the Systems Engineering Research Center (SERC) under Contract H98230-08-D-0171 and the National Security Agency (NSA) under lablet contract #H98230-14-C-0140.

We thank Thorsten Berger for providing us a large corpus of Android apps and Alexander von Rhein for providing us the script to download apps from Play Store. We also thank Damien Octeau for providing support in using the IC3 tool.

[14] http://www.omg.org/gettingstarted/omg_idl.htm

11. REFERENCES

[1] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, 2002.

[2] C. Artho, K. Suzaki, R. D. Cosmo, R. Treinen, and S. Zacchiroli. Why do software packages conflict? In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 141–150, 2012.

[3] T. Berger, R.-H. Pfeiffer, R. Tartler, S. Dienst, K. Czarnecki, A. Wąsowski, and S. She. Variability mechanisms in software ecosystems. *Information and Software Technology*, 56(11):1520–1535, November 2014.

[4] M. Blume and A. W. Appel. Hierarchical modularity. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 21(4):813–847, 1999.

[5] C. Bogart, C. Kästner, and J. Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *Proceedings of the ASE Workshop on Software Support for Collaborative and Global Software Engineering*, 2015.

[6] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[7] R. D. Cosmo, D. D. Ruscio, P. Pelliccione, A. Pierantonio, and S. Zacchiroli. Supporting software evolution in component-based FOSS systems. *Sci. Comput. Program.*, 76(12):1144–1160, 2011.

[8] B. E. Cossette and R. J. Walker. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 55:1–55:11, 2012.

[9] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6, 2010.

[10] L. Eshkevari, G. Antoniol, J. R. Cordy, and M. Di Penta. Identifying and locating interference issues in PHP applications: The case of WordPress. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 157–167, 2014.

[11] M. Greiler, A. van Deursen, and M.-A. Storey. Test confessions: A study of testing practices for plug-in systems. In *Proceedings of the 34th International Conference on Software Engineering*, pages 244–254, 2012.

[12] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *Proceedings of the 37th International Conference on Software Engineering*, pages 100–110, May 2015.

[13] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 1978.

[14] D. Hou and X. Yao. Exploring the intent behind API evolution: A case study. In *Proc. Working Conf. Reverse Engineering*, pages 131–140, 2011.

[15] R. Kazman and H.-M. Chen. The metropolis model and its implications for the engineering of software ecosystems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pages 187–190, 2010.

[16] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6, 2014.

[17] M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept 1980.

[18] M. Lehman and J. F. Ramil. Software evolution in the age of component-based software engineering. *IEEE Explore*, pages 249–255, 2002.

[19] X. Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–122, 1994.

[20] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, , and P. Mcdaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering*, pages 280–291, 2015.

[21] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. API change and fault proneness: A threat to the success of Android apps. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering*, pages 477–487, 2013.

[22] M. Linares-Vásquez, G. Bavota, M. D. Penta, R. Oliveto, and D. Poshyvanyk. How do API changes trigger Stackoverflow discussions? A study on the Android sdk. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 83–94, 2014.

[23] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer. An empirical study of the robustness of inter-component communication in Android. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security*, pages 73–84, 2013.

[24] K. Manikas and K. M. Hansen. Software ecosystems – a systematic literature review. *The Journal of Systems and Software*, 86(5):1294–1306, May 2013.

[25] T. McDonnell, B. Ray, and M. Kim. An empirical study of API stability and adoption in the Android ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, pages 70–79, 2013.

[26] T. Mens and S. Demeyer, editors. *Software Evolution*. Springer, 2008.

[27] E. Newcomer. *Understanding Web Services: XML, WSDL, SOAP, and UDDI*. Addison-Wesley Professional, 1st edition, 2002.

[28] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 907–918, 2014.

[29] A. Nhlabatsi, R. Laney, and B. Nuseibeh. Feature

interaction: The security threat from within software systems. *Progress in Informatics*, pages 75–89, 2008.

[30] D. Octeau, D. Luchaup, M. Dering, S. Jha, , and P. McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering*, pages 77–88, 2015.

[31] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security*, pages 543–558, 2013.

[32] ROS.org. Ros documentation. http://wiki.ros.org, 2015.

[33] R. Strniša, P. Sewell, and M. Parkinson. The Java module system: Core design and semantic definition. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications*, pages 499–514, New York, 2007. ACM Press.

[34] A. von Rhein, T. Berger, N. S. Johansson, M. M. Hardø, and S. Apel. Lifting inter-app data-flow analysis to large app sets. Technical Report MIP-1504, Department of Informatics and Mathematics, University of Passau, Sept. 2015.

[35] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341, 2014.

[36] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. Appscope: Application energy metering framework for Android smartphones using kernel activity monitoring. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, 2012.