2013 IEEE International Conference on Software Maintenance

An Empirical Study of API Stability and Adoption in the Android Ecosystem

Tyler McDonnell, Baishakhi Ray, Miryung Kim
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX, USA
Email: tscottmcdonnell@gmail.com, rayb@utexas.edu, miryung@ece.utexas.edu

***Abstract*—When APIs evolve, clients make corresponding changes to their applications to utilize new or updated APIs. Despite the benefits of new or updated APIs, developers are often slow to adopt the new APIs. As a first step toward understanding the impact of API evolution on software ecosystems, we conduct an in-depth case study of the co-evolution behavior of Android API and dependent applications using the version history data found in github.**

Our study confirms that Android is evolving fast at a rate of 115 API updates per month on average. Client adoption, however, is not catching up with the pace of API evolution. About 28% of API references in client applications are outdated with a median lagging time of 16 months. 22% of outdated API usages eventually upgrade to use newer API versions, but the propagation time is about 14 months, much slower than the average API release interval (3 months). Fast evolving APIs are used more by clients than slow evolving APIs but the average time taken to adopt new versions is longer for fast evolving APIs. Further, API usage adaptation code is more defect prone than the one without API usage adaptation. This may indicate that developers avoid API instability.

I. Introduction

Over the course of the past few years, the mobile application arena has exploded with the dissemination of affordable and powerful smart phones. As of 2012, the Apple App Store and Google Play Store boast a combined 1.5 million available apps and 55 billion app downloads worldwide [27]. Google and Apple support mobile app development with their own operating systems and associated Application Programming Interfaces (APIs). These APIs give developers access to features like location services, wi-fi connections, bluetooth functionality, and graphics.

When APIs evolve to accommodate new feature requests, to fix bugs, to meet new standards, and to provide higher performance, client applications often need to make corresponding changes to use new or updated APIs. Despite the benefits of new or improved APIs, developer adoption is often slow among client applications. For example, the Android Operating System is evolving fast, yet API adoption is slow and the consumer pool is fragmented by the Android version numbers [5].

Though many techniques have been proposed to ease library migration and to address API version incompatibilities, API evolution and its associated ripple effect throughout software ecosystems are still under-studied. For example, Robbes et al. [21] studied how client applications react to API evolution in libraries or frameworks, but the study was confined to the issue of API deprecation in Smalltalk.

As a first step towards understanding the impact of API evolution on developer adoption, we conduct an in-depth case study of the Android API and dependent applications. Using the version history data of Android applications found in github and the API evolution data derived from Android OS documentation pages, we quantify their co-evolution behavior. We analyze the average time between API updates and record the number of method and field changes in each Android version. We also track changes in each major feature of Android. On the side of client applications, we calculate the percentage of Android API method calls and field references and categorize them by the API version number. By comparing the commit time of an API reference in client code against the release time of newer APIs, we identify outdated API usages and measure the *lagging time*—how far existing API references are lagging behind newly released APIs. We also measure the *propagation time* —the time taken for the client code to adopt new API usages. Then we correlate these adoption statistics with the frequency and location of evolving APIs in Android OS.

By characterizing the co-evolution behavior of APIs and dependent applications, we address the following research questions. Our findings are summarized as follows:

- **How fast does the Android API change, and which parts change the most?** Android APIs are evolving at the rate of 115 API updates per month on average. APIs related to hardware, user interface, and web are evolving much faster than others.
- **How dependent is client code on Android APIs, and how long does it take to adopt new APIs?** Around 25% of all method and field references in the client code use the Android APIs. However, application developers are hesitant to adopt new APIs. On average, 28% of Android API calls are lagging behind the latest released version. 22% of outdated APIs eventually upgrade to use newer APIs; nevertheless it takes a considerable amount of time, 14 months, on average.
- **What is the relationship between API stability, usage, adoption, and bugs?** Fast evolving APIs are used more by clients than slow evolving APIs. However, the

pace of client update is slower for fast evolving APIs. Files which are changed to use new APIs are more defect prone than files without API usage adaptation. This may imply that developers avoid frequent upgrades to unstable or rapidly evolving APIs.

To the best of our knowledge, we are the first to quantify the co-evolution behavior of Android and mobile applications and to confirm that client adoption is not keeping pace with API evolution. We are the first to find that API updates are more defect prone than other types of changes by investigating the relationship between API instability and bugs in client code as opposed to a library. Our study shows that fast-evolving APIs are used more and adopted more, but the time taken for API adoption is longer. Though many tools exist to automate API usage updates in client code, these tools are inadequate for promoting adoption alone as various stakeholders affect the process of API adoption. We call for further studies on how to promote API adoption and ultimately facilitate the growth of software ecosystems.

II. Related work

Empirical Studies of API Evolution. In this paper, we seek to understand developer response to evolving APIs. Several studies analyze different software ecosystems and attempt to assess the *ripple effect* that API changes may have on client applications. Dig and Johnson found that 80% of the code changes that break client-side code are API refactorings [10]. Similarly, Xing and Stroulia studied Eclipse evolution history and found that 70% of structural changes are due to refactorings and existing IDEs lack support for complex refactorings [28]. In our study of API evolution, we examine the relationship between API stability and the degree of adoption measured in propagation and lagging time, which have not been investigated before in the above studies. Hou and Yao study the Java API documentation and find that a stable architecture played an important role in supporting the smooth evolution of the AWT/Swing API [15].

In a large scale study of the Smalltalk development communities, Robbes et al. found that only 14% of deprecated methods produce non-trivial API change effects in at least one client-side project; however, these effects vary greatly in magnitude. On average, a single API deprecation resulted in 5 broken projects, while the largest caused 79 projects and 132 packages to break [21]. In contrast to Robbes et al., our study is not limited to API deprecation and we focus on applications written in Java as opposed to Smalltalk. The mobile software arena may also differ from other applications by placing the burden on developers to support users running a wide variety of devices and different OS versions.

Kim et al. investigate the relationship between API refactorings and bugs and find that the number of bug fixes increases after API refactorings [16]. Weißgerber and Diehl also find that API refactorings often occur together with other types of changes and that API refactorings are followed by an increasing number of bugs [26]. These studies investigate the relationship between API refactorings and bugs in libraries only, as opposed to bugs in clients.

Yau et al. [29] and Black [7] investigate the ripple effects of evolving software, but these studies focus on a single system, as opposed to the impact of evolving APIs on clients. In this paper, we investigate how mobile applications react to API changes in the Android *ecosystem*, following Lungu et al.'s definition—*a collection of software projects which are developed and co-evolve in the same environment* [18].

Techniques for Easing API Migration. Several techniques can help programmers deal with broken code as a result of API evolution. Henkel and Diwan and Ekman and Asklund record API refactorings performed in an IDE in order to replay them in the client applications [13], [14]. Dig et al. present a refactoring-aware version control system to account for refactoring edits during the version merging process [11]. Chow and Notkin suggest how to adapt client applications using API usage adaptation rules written by developers [8]. Dig et al. adopt a proactive approach and create a layer between clients and each updated library version [12]. This approach has the advantage of preventing broken code with no client-side effort, but it can discourage the use of new functionality of upgraded APIs.

Other techniques recommend API replacement using various types of underlying analyses: lexical comparison of method signatures syntactic and semantic similarity of APIs, shingles analysis, analysis of how a library's internal API usage changes between versions, analysis of code comments and release documents, source code implementation details, analysis of how other developers adapted their code, and combination of method signatures and call usages. A survey of techniques for easing API migration is found elsewhere [9], [17]. Cossette and Walker found that, while most broken code may be mended using one or more of these techniques, each is ineffective when used in isolation [9].

Studies on Android Applications. Shabtai et al. is the first to conduct a formal study of Android Packages files (APK) [24]. They apply machine learning techniques to classify applications into two categories: tools vs. games. Syer et al. [25] compare the source code size, churn, and dependency characteristics of mobile applications for the Android platform against those for the Blackberry platform. Ruiz et al. investigate the extent of reuse in the Android Market using Software Bertillonage techniques to track code across mobile applications [22]. Sanz et al. [23] detect malicious Android applications in the Android market. Our study differs from these studies by investigating the impact of evolving APIs on adoption. By analyzing bug reports and developers' discussion, Pathak et al. [20] find that 20% of overall energy related bugs in Android occur after an OS update. This may explain our study finding that the number of bugs increases in Android applications following an API update.

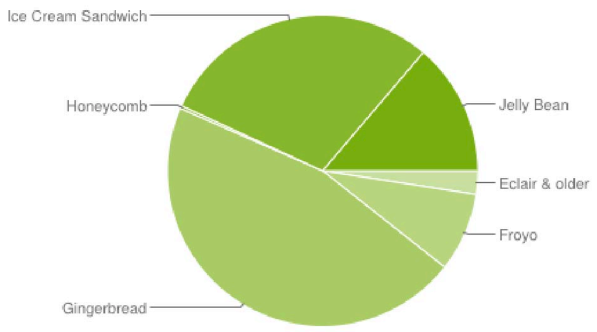
III. Study Method

Section III-A describes the Google Android API, its evolution history, and developer pool. Section III-B describes the client applications we selected as subjects.

A. Android API

Android is an open source Linux-based operating system owned by Google and designed for touchscreen mobile devices such as smart-phones and tablet computers.

The rapid commercial growth of the mobile sector is coupled with similarly fast-paced hardware and software evolution. Google released the first Android version (Android 1.0, API level 1) on September 23rd 2008. Since then a new version is released approximately in every 3 months, till the release of the current version, (Android 4.2, API level 17) on November 13th 2012. Figure 1 shows the API version history, including release dates, version numbers, associated API levels, and codenames [4].



Version	Codename	API	Dist.	Release Date
1.0	None	1	*	Sep 23, 2008
1.1	None	2	*	Feb 9, 2009
1.5	Cupcake	3	*	Apr 30, 2009
1.6	Donut	4	0.2%	Sep 15, 2009
2.0	Eclair	5	*	Oct 26, 2009
2.0.1	Eclair	6	*	Dec 3, 2009
2.1	Eclair	7	8.1%	Jan 12, 2010
2.2	Froyo	8	8.1%	May 20, 2010
2.3-2.3.2	Gingerbread	9	0.2%	Dec 6, 2010
2.3.3-2.3.7	Gingerbread	10	45.4%	Feb 9, 2011
3.0	Honeycomb	11	*	Feb 22, 2011
3.1	Honeycomb	12	0.3%	May 10, 2011
3.2	Honeycomb	13	1.0%	Jul 15, 2011
4.0-4.0.2	Honeycomb	14	*	Oct 19, 2011
4.0.3-4.0.4	Ice Cream Sandwich	15	29.0%	Dec 16, 2011
4.1	Jelly Bean	16	12.2%	Jul 9, 2012
4.2	Jelly Bean	17	1.4%	Nov 13, 2012

data provided by Google
* indicates that no distribution data is available

Figure 1. Client API Usage

Because the Android operating system runs on a wide variety of different devices, the consumer pool is fragmented by Android version numbers. Older mobile devices often ship with earlier versions of the Android OS and may be incapable of updating to the most recent Android API due to hardware limitations. Figure 1 shows the current breakdown of Android version usage based on devices that accessed Google Play within a 14-day period in January 2013 [4]. Notably, this distribution shows either a hesitancy towards newer versions or at least a slow adoption trend, with 50% of users running a version released 20 or more months prior to the current version. Many analysts note how little change they have seen in the market share of the dominant Android version in the past 12 months and how this fragmentation differs from Apple's iOS [6].

For this study, we correlate changes in client applications with changes in Android OS. We build a data structure to store the Android API version history. We analyze the `api-versions.xml` file and the `apidiff` directory that Google provides to developers along with the Android Software Development Kit (SDK). The `api-versions.xml` file provides a complete listing of all Android classes, methods, and fields available in API version 1. The `apidiff` directory is a set of `html` files, cataloging changes to any classes, methods, or fields from the previous API version.

B. Study Subjects

We use *active* open source Android applications with the following traits in github: over 1000 commits, 5 or more authors, 100 or more line changes per commit, and at least one commit made in 2012.

CP Congress Tracker is an app that allows users to manage a personal schedule, locate opinion leaders, and provide general coverage of related congressional events. **Apollo Music Player** is a customizable lightweight Android music app. **Cyanogen** is a set of multimedia and user interface suites. It allows users to place widgets like analog clocks on the home screen of their phone. **Google Play Analytics** allows users access real-time Google Analytics profiles. **LastFM** is a music listening and sharing application. **mp3Tunes** allows users to access and listen to the songs in iTunes. **OneBusAway** is a mobile app that provides real-time arrival information for Seattle area buses. **ownCloud** allows users to access and share files stored in the cloud. **RedPhone** is an app for making secure calls by providing end-to-end encryption. **XMBCremote** is a full featured remote control software for the XMBC media center. Table I shows the last updated time, the number of revisions, change rate, the number of authors, and code size.

IV. Study Results

Section IV-A presents the extent and characteristics of Android API evolution. Section IV-B investigates how fast client code is adapting the updated APIs. Section IV-C analyzes the impact of API evolution on client code.

A. Characteristics of Android API Evolution

RQ1. How fast do Android APIs evolve? We first identify added, changed, and removed APIs for each Android version. When a new API version is released, Google provides an html file documenting all API changes in each release [4]. We built a tool that extracts API change information from the html file and stores the API Version history data. We define a changed class as one that is either new in the particular version or has at least one changed, added, or removed API method or field. Similarly, changed methods and fields are the ones with a modified signature since the previous version. Removed methods and fields are those that existed in the previous version but no longer do in the current version. Table II shows the extent of API evolution at the class, method, and field granularity.

To measure the rate of Android API evolution, we compute how many APIs are updated in each month on average.

$$avg.\ update\ rate = \frac{\sum_{releases} \#\ API\ updates}{\#\ months}$$

In each month, 44 methods are changed, 11 methods are added, 51 fields are changed, 9 fields are added, and less than one method or field is removed on average. Android is constantly evolving to include more method and field variables, with existing methods and fields frequently changed to add new functionality. However, removal of existing functionalities is rare.

RQ2. What functions of Android API are updated most? We investigate the areas of the Android API that are updated most frequently. From Android packages, we select certain *keywords* to characterize API features. For example, the taxon *text* is drawn from all packages relevant to rendering or tracking text on an Android device: `android.text.format`, `android.text.method`, `android.text.style`, and `android.text.util`. We create 25 taxa using various keywords such as: animation, bluetooth, database, graphics, io, os, security, and text. We then categorize each new, changed, or removed API method and field for each taxon. Table III shows which taxa are updated most frequently.

Table I
CHARACTERISTICS OF CLIENT MOBILE APPS

Apps	Updated	Rev	Rev/mo	Author	LOC
Congress Tracker	04-15-2013	1359	25.6	7	13349
Apollo M	03-24-2013	9	0.4	1	15783
Cyanogen	01-10-2011	109	2.3	20	28972
Google A	03-12-2013	926	77.1	23	52932
LastFM	03-03-2013	212	8.2	7	9771
mp3Tunes	02-17-2013	104	2.2	1	9608
OneBusAway	03-09-2013	497	33.1	5	51784
ownCloud	04-12-2013	665	55.4	12	25109
RedPhone	03-23-2013	116	4.8	5	21315
XMBCremote	04-05-2013	928	19.3	24	92893

Table II
API CHANGES IN ANDROID PER VERSION AND EVOLUTION RATES

API	Release	Class	Methods			Fields		
Version	Date	Δ	Δ	+	-	Δ	+	-
3	Apr 30, 2009	246	368	60	0	296	68	0
4	Sep 15, 2009	128	70	41	1	208	27	0
5	Oct 26, 2009	187	199	64	0	234	205	0
6	Dec 3, 2009	37	0	2	0	7	1	0
7	Jan 12, 2010	61	52	2	0	22	3	0
8	May 20, 2010	191	200	38	1	195	23	0
9	Dec 6, 2010	244	348	42	9	141	11	0
10	Feb 9, 2011	46	7	0	0	10	0	0
11	Feb 22, 2011	263	416	95	7	619	36	0
12	May 10, 2011	118	73	27	1	87	9	0
13	Jul 15, 2011	69	22	11	0	68	1	0
14	Oct 19, 2011	269	271	98	8	405	34	0
15	Dec 16, 2011	84	25	3	0	38	2	0
Min		37	0	0	0	7	0	0
Max		269	416	98	9	619	205	0
Mean		149	158	37	2	179	32	0
Rate (Total update/month)		**42**	**44**	**11**	<1	**51**	**9**	**0**

Table III
API CHANGE DISTRIBUTION PER TAXON (FEATURE)

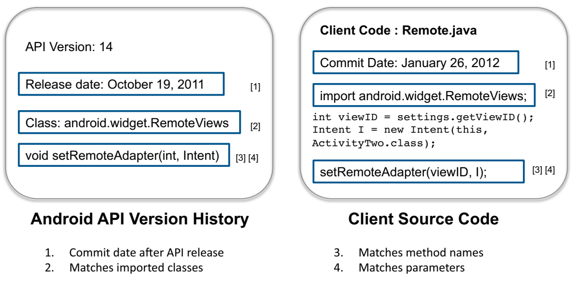
Taxon	Total Updated Versions	Total Updated APIs	Avg. Changes Per API Release	Avg Update Interval (Month)
animation	7	37	5	5.4
appwidget	3	12	4	12.7
bluetooth	5	9	2	7.6
content	10	179	18	3.8
database	6	100	17	6.3
gest	1	3	3	38.0
graphics	10	84	8	3.8
hardware	10	121	12	3.8
io	2	18	9	19.0
location	4	38	10	9.5
media	8	93	12	4.8
net	8	87	11	4.8
opengl	5	10	2	7.6
os	11	94	9	3.5
rtp	0	0	0	
security	2	25	13	19.0
sip	1	2	2	38.0
support	0	0	0	
telephony	5	49	10	7.6
test	8	70	9	4.8
text	9	147	16	4.2
util	6	180	30	6.3
view	12	546	46	3.2
webkit	10	172	17	3.8
wifi	4	14	4	9.5

We find that the most frequently evolving packages (the lowest average time between API changes) include `content`, `graphics`, `hardware`, `os`, `view`, and `webkit`—each updated in 10 or more of the 14 API releases under consideration. We also find that `content`, `hardware`, `text`, `util`, `view`, and `webkit` have more than 100 API changes since the original Android release.

The high frequency of API updates related to hardware, graphics, and views may be due to the Android hardware fragmentation. In contrast to iOS, which supports five unique devices, there are at least 170 Android devices. Google may rapidly update the hardware and graphics APIs to support widely-varying hardware features.

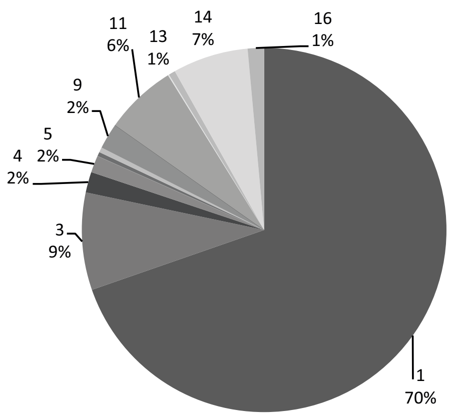
B. Characteristics of API adoption by the client programs

This section investigates how client programs respond to API evolution. We study the degree of dependence that client applications have on Android APIs and how fast clients are adopting new or updated APIs.


API Version: 14
Release date: October 19, 2011
[1]
Class: android.widget.RemoteViews
[2]
void setRemoteAdapter(int, Intent)
[3] [4]
Android API Version History
Client Code : Remote.java
Commit Date: January 26, 2012
[1]
import android.widget.RemoteViews;
[2]
setRemoteAdapter(viewID, I);
[3] [4]
Client Source Code
1. Commit date after API release
2. Matches imported classes
3. Matches method names
4. Matches parameters

```
int viewID = settings.getViewID();
Intent I = new Intent(this,
ActivityTwo.class);
```

Figure 2. Identifying Android References in Client Source Code



Client Applications	Android API	Total API	% Android API	Unique Android API
Congress Tracker	1007	3396	30%	82
Apollo Music	1332	3820	35%	155
Cyanogen	1439	5992	24%	144
Google A	3164	12145	26%	336
LastFM	371	2122	16%	64
mp3Tunes	510	2275	22%	101
OneBusAway	2416	10932	22%	297
ownCloud	1838	6132	30%	194
RedPhone	830	4303	19%	160
XMBCremote	3209	14626	22%	275

Figure 3. Degree of Android API dependence of client code

RQ3. How dependent is client code on Android APIs? We analyze client application source code to measure the degree of dependence on Android APIs. We identify all Android methods and fields references using a syntax-based lexical search on Java source files. By analyzing the *import* statements, we detect the Android classes referenced in each Java file. For each referenced API method call or field access, we search through our Android API Version History data structure (see Section IV-A) to find a corresponding API declaration and its version. We match an API invocation with a corresponding API declaration based on a method name and the number of parameters. We detect API usage updates by monitoring changes to the used method name, the number of arguments, and argument names. We also use the commit date of a source file to determine the most recent available API version. For example, Figure 2 shows an Android API method invocation `setRemoteAdapter(int, Intent)` in client code. When scanning the client source file, we find an entry in the Android API Version History data structure of method `setRemoteAdapter`. By matching the number of parameters, the release date of the API entry, the commit date of client code, and the list of imported classes in the source file, we infer that the client code is using the API version 14 for method `setRemoteAdapter(int, Intent)`.

By measuring the proportion of Android API method calls and field references out of all references, we investigate how dependent client apps are on Android APIs (see Figure 3). Approximately 25% of all method and field references in client code are about Android APIs. Around 80% of the references in the most recent version of client code refer to Android API Version 1 or Version 3, released in September 2008 and April 2009 respectively. These results show that though mobile apps are heavily dependent on Android OS and its functionality, developers are hesitant to embrace or fully utilize more recent API features.

RQ4. What is the lag time between client code and the most recent Android API? We detect the *lag time* between a client API reference (i.e., API method calls) and its most recent available version. An API method invocation in client code is considered to be *lagging* if a more recent version of the method is available at the time of its commit. We define the *lag time* of outdated API usage as the number of months elapsed between the release of the new version and the commit time of the outdated API usage code. For example, Figure 4 shows `setbutton2(charSequence)` is deprecated between API version 4 and 7. In client code, developers use the deprecated method at a later date, on December 20th 2009. We consider this method reference is *lagging* because the method was deprecated prior to the client code commit. The **lag time** in this case is approximately two months, the time difference between the client code commit on December 20th 2009 and the deprecation in API Version 7 on October 26th 2009.

We measure the number of outdated API calls and their lagging time. This analysis is done on a `git` commit granularity. For each API invocation in each commit, we first identify the used API version by comparing the method signature of the API call in client code with our Android API Data Structure. We then retrieve the most recent API version of that method available at the time of commit. Finally, by comparing the commit date and the release date of its

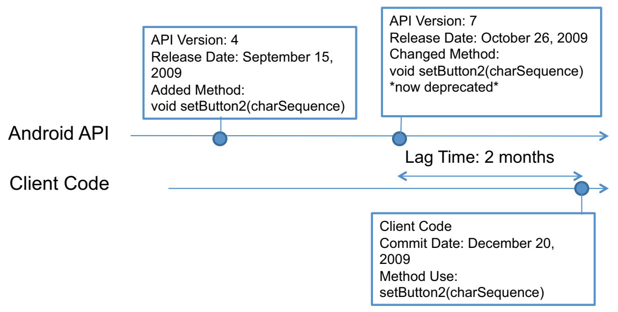


Figure 4. Lag Time Example

updated version, we compute the lagging time. Table IV summarizes the results. At any point in time, on average, 28% of Android method calls are out-of-date and lagging behind the most recent available Android API version. The percentage of outdated API usage varies from a minimum value of 11% to a maximum value of 43% on average.

Table IV
LAG TIME STATISTICS

Apps	Lag (# Methods)			# Affected Files		
	Max	Avg	Min	Max	Avg	Min
Congress T	516 (50%)	216 (18%)	0 (0%)	81	64	0
Apollo M	968 (72%)	964 (72%)	961(72%)	64	64	64
Cyanogen	256 (17%)	171 (12%)	0 (0%)	35	20	0
Google A	1784 (46%)	1409 (37%)	0 (0%)	134	86	0
LastFM	291 (70%)	181 (43%)	0 (0%)	47	28	0
mp3Tunes	47 (8%)	26 (5%)	4 (1%)	13	8	4
OneBusAway	19 (4%)	14 (3%)	0 (0%)	4	2	0
ownCloud	1488 (52%)	489 (18%)	4 (<1%)	171	121	2
RedPhone	547 (48%)	498 (43%)	414 (35%)	82	72	69
XMBCremote	1421 (41%)	537 (15%)	0 (0%)	238	123	0
Mean	777 (43%)	451 (28%)	138 (11%)	87	60	14

We combine the lagging time results across all subject apps to produce a cumulative distribution of lag time in Figure 5. 50% of all outdated API references are lagging behind the most recent available API by 16 or more months.

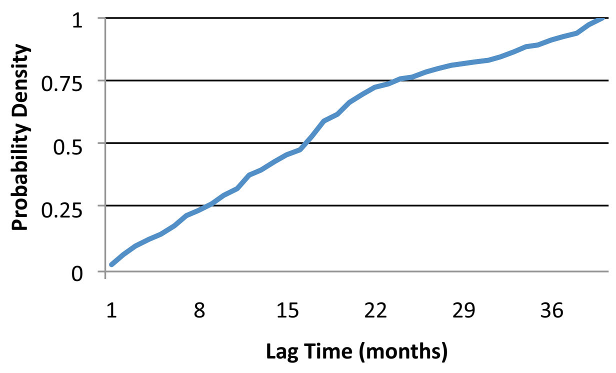


Figure 5. Cumulative Distribution of Lag Time (CDF)

The results suggest that developers do not quickly adopt new APIs. They keep the outdated API references, avoiding the instability of newer APIs and the work that comes with API upgrade.

RQ5. How long does it take for API changes to propagate throughout the Android ecosystem? We measure how long it takes for clients to adopt new API usages once a new or updated API becomes available. When a method is eventually updated to a newer API version in client code, we measure its *propagation time*—time difference in months between the API release and the client adaptation timing when the updated usage is committed in the client repository. Figure 6 illustrates this concept by comparing the parallel evolution of Android and a client project. In the Android development time line, the signature of `getMethod` is altered in API version 9 on December 6th 2010 to include an additional `Class` parameter. Client code committed on March 8th 2011 changes the usage of `getMethod` to match the updated API version 9 signature. In this example, the *propagation time* is three months, the time difference between the updated API usage on March 8th 2011 and the release of API version 9 on December 6th 2010.

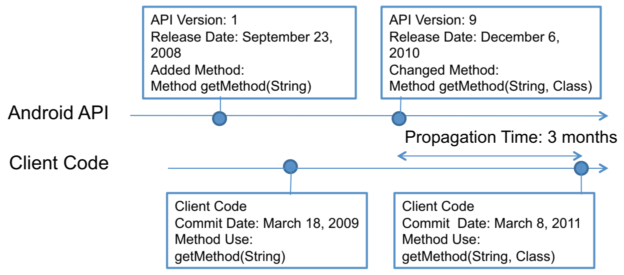


Figure 6. Propagation Time Example

Figure 7 shows the results of propagation time analysis. We inspect each commit patch of the client projects to look for method calls updated to new APIs. For each updated method, we record the propagation time in months. Figure 7 represents the distribution of propagation times across all subject applications in the form of a cumulative distribution plot. The mean propagation time is 14 months (or almost five Android releases) and 50% of all API usage updates occur within approximately 14 months of the associated API release. Outdated API usages eventually upgrade to use newer APIs, but at a much slower pace than the rate of API evolution.

C. Interplay between Android API evolution and client adoption.

This section investigates the relationship between API stability, usage, adoption, and bugs.

RQ6. What is the relationship between API stability and adoption? To understand how API stability affects

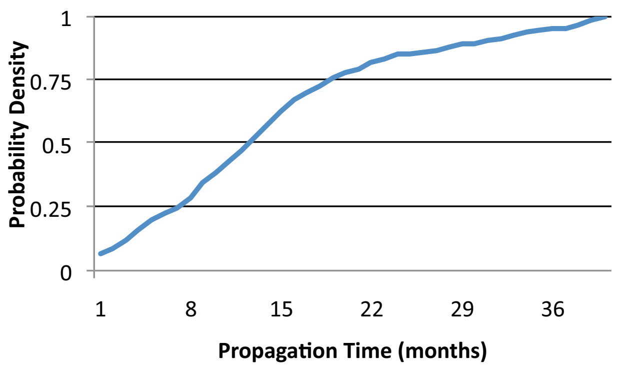


Figure 7. Propagation Time of Methods in Client Code (CDF)

adoption, we measure the Spearman rank correlation between the API evolution rate and adoption measures. For each taxon in Table III, we use the average API update interval (Column Avg Update Interval in Table III), the percentage of total API usages that taxon accounts for (API usage), and the number of API references that were upgraded to newer API versions in client code (propagation count). Table VI summarizes the results. The Spearman correlation between API update interval and API usage is -0.47 with a p-value of 0.01757 (See Table VI). A negative correlation value indicates that fast evolving APIs are used more by clients and this trend is statistically significant. The left graph of Figure 8 represents the average API update interval and the API usage percentage for each taxon.

The Spearman correlation between the average API update interval and the propagation count is -0.707 with a p-value 0.0001113). A negative correlation suggests that clients upgrade to faster evolving APIs more frequently. The right side of Figure 8 shows the average API update interval and the number of API usage propagations. These results indicate that faster evolving APIs are used and adopted more by clients.

RQ7. What is the relationship between API usage and adoption? To understand the relationship between API usage and adoption, we measure the Spearman correlation between API usage and the average propagation time per taxon (see Table V). When computing the correlation, we remove all taxa whose propagation count is zero. The correlation is 0.6966 with p-value 2.72E-03, indicating that APIs that are used more often have higher propagation times. In conjunction with RQ6's results, this implies that the pace of client updates is slower for widely used, faster evolving APIs and that developers avoid frequent updates to unstable APIs.

We found a positive correlation of 0.844 between API usage and propagation count with p-value 1.93E-05 (see also Figure 9 for the graph on API usage % and propagation count). The more an API is used, the higher its number of

Table VII
SPEARMAN RANK CORRELATION BETWEEN BUG FIXES AND API UPDATES

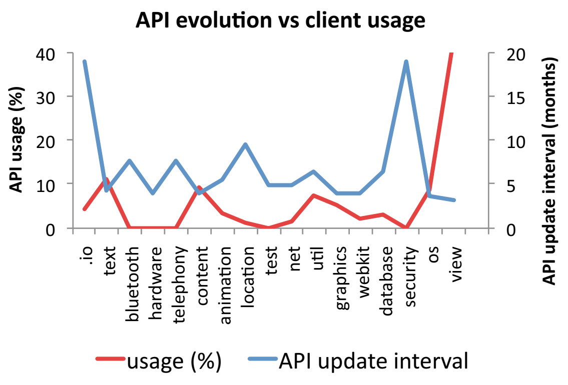
Client Application		Correlation with bugs	p-value
Congress T	Total CLOC	0.39	1.33E-13
	API Update CLOC	0.56	2.20E-16
	Non API Update CLOC	0.39	1.96E-13
OneBusAway	Total CLOC	0.26	1.06E-07
	API Update CLOC	0.46	2.20E-16
	Non API Update CLOC	0.25	2.28E-07
RedPhone	Total CLOC	0.23	1.44E-03
	API Update CLOC	0.24	1.13E-03
	Non API Update CLOC	0.23	1.48E-03
XMBCremote	Total CLOC	0.34	2.20E-16
	API Update CLOC	0.62	2.20E-16
	Non API Update CLOC	0.33	2.20E-16
Google Analytic	Total CLOC	0.36	1.92E-11
	API Update CLOC	0.54	2.20E-16
	Non API Update CLOC	0.31	5.83E-09
OwnCloud	Total CLOC	0.43	6.113E-16
	API Update CLOC	0.55	2.2E-16
	Non API Update CLOC	0.42	2.81E-15
Cyanogen	Total CLOC	0.58	8.53E-08
	API Update CLOC	0.63	3.749E-09
	Non API Update CLOC	0.58	1.035E-07
LastFM	Total CLOC	0.42	1.10E-07
	API Update CLOC	0.37	5.18E-06
	Non API Update CLOC	0.43	1.04E-07

client code updates. In other words, highly used taxa are adopted more frequently.

RQ8. What is the relationship between API updates and bugs in client code? To investigate how API updates affect the likelihood of defects in client code, we analyze the correlation between the number of bugs and the amount of lines changed for the purpose of upgrading to newer APIs. By analyzing version history, we identify java files changed in each commit. For those files, we measure the number of added lines, the number of changed lines for the purpose of upgrading to newer APIs, and the number of changed lines not related to API updates. Next, using a heuristic similar to Mockus and Votta [19], we identify the number of bug fixes by searching for commit messages with the keywords: `bug`, `error`, `fix`, and `solve`.

We then calculate the Spearman correlation between bug fix CLOC and API upgrade CLOC in client code at the file granularity [30]. We also measure how bugs are correlated with total CLOC and non-API upgrade CLOC respectively. Table VII shows the results. The correlation between bug fixes and API updates is stronger than the correlation between bug fixes and non-API updates in all applications except LastFM.

These results show that, in general, the files with more API updates are more prone to bugs. The stronger correlation



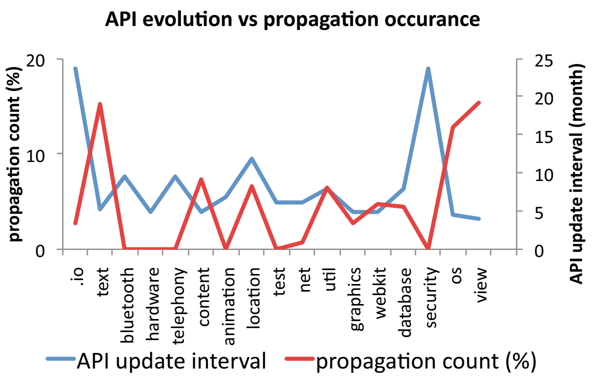


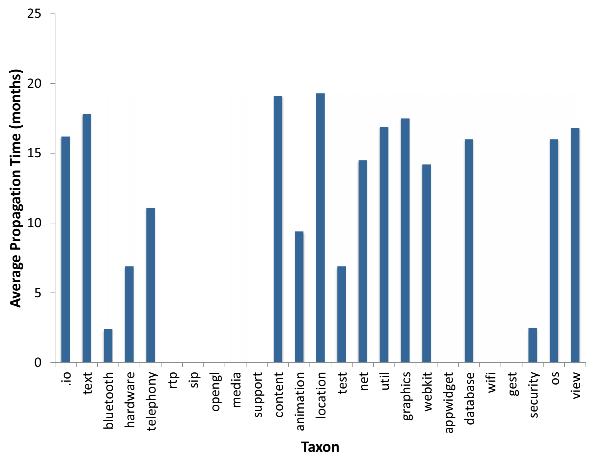
Figure 8. Taxa API update vs. client adoptions

Table V
CORRELATION BETWEEN API PROPAGATION AND API USAGE

		correlation	p-value
# API usage (%)	propagation time	0.6966134	2.72E-03
# API usage (%)	propagation count	0.8441176	1.93E-05

Table VI
CORRELATION BETWEEN API EVOLUTION AND CLIENT ADOPTION

		correlation	p-value
avg API update interval	API usage (%)	-0.4706808	0.01757
avg API update interval	propagation count	-0.7072448	0.0001113



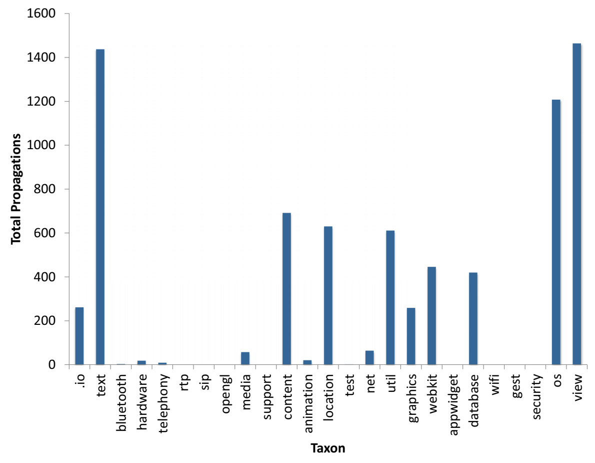


Figure 9. Average Propagation Time and Total Propagation Count for Each Taxon

between API update and bug-fix may explain the slower adoption of new APIs—developers may be skeptic about API adoption as it may introduce bugs. In fact, many developers notice that the API implementation on the Android OS side is often buggy when released. For example, several major bugs related to random rebooting and excessive battery drainage were reported regarding Android 4.2 (Jelly Bean) release [1]. Because of these bugs, developers were hesitant to adopt new APIs or frustrated with their attempts at adoption, some claiming that the release was *the most buggy update since Honeycomb* and they were *definitely expecting an update pretty soon* [1].

V. THREATS TO VALIDITY

Regarding threats to *construct validity*, we use a syntax-based lexical search to identify Android API references in client code, and we match an API reference with a corresponding API declaration based on the API method name and the number of arguments without considering argument types. For example, consider an API method declaration is updated from `void foo(char, int)` to `void foo(char, char)` on a library side. If nothing is altered at a client call site except the type of the second argument, we cannot detect the change.

Because our method detects API usage change in client code by keeping track of the used API method name, the number of arguments, and argument names, our method could accidentally detect an API usage update when a method invocation is changed from `foo(varA, varB)` to `foo(varA, varC)` even though `varC` is a simply renamed variable of `varB`. While calculating the propagation time and lag time of API references, our method considers API method invocations only, not changes to how API fields were read or used.

Additionally, it is possible for an application to support many different API versions simultaneously [2]. Developers can use version specific APIs inside *if* and *switch* blocks. At runtime, using a Android Build class the API version of a device can be retrieved and appropriate conditional blocks can be executed. Our method of detecting and logging lagging methods does not take into account such multi-version API support.

In terms of threats to *internal validity*, our study presents the correlation between API usage, adoption, and bugs, but not causation. Furthermore, regarding outdated API usages, it is possible that clients are purposely leaving outdated API usages in the codebase to account for the distribution of Android user base. In fact, we find that the most number of propagations are about upgrading to API version 10 (Gingerbread), which currently has the highest market share. The average propagation time to versions up to Gingerbread is higher than that of later versions. This may suggest that developers may eventually gravitate to the versions with a large number of users. Similarly, a high correlation between API updates and bug fixes may be caused by factors beyond our study scope such as test coverage, expertise, etc.

Regarding *external validity*, we investigate Android and ten open source mobile applications found in github, and the results may not generalize to a broader set of mobile applications. Because we chose projects with high activity statistics from different application categories, we believe that our results provide valuable insights on the co-evolution behavior of API and dependent applications.

VI. Conclusion and Future Work

In this paper, we perform an empirical study on the co-evolution of Android OS and its clients. Android is evolving fast with an average of 115 API updates per month. Although client applications are heavily dependent on Android, developers seem hesitant to embrace unstable, fast-evolving APIs quickly: 28% of Android references in client code are out-of-date with a median lag time of 16 months. Approximately 22% of these outdated references are eventually adapted to use newer APIs, but the average propagation time is 14 months, or almost 5 Android API version updates.

Furthermore, we study correlations between API usage, API evolution rate, the time taken for API adoption, and the number of bugs in client code. The APIs that clients use most are the ones Google update most frequently. These same APIs have the higher number of propagations, but with the greater hesitancy (i.e., longer propagation time). Connecting these results with our finding on defect-proneness of API usage adaptation, we believe that developers are hesitant to quickly adopt new, unstable APIs, but eventually tend to migrate their code to keep up with the mass of the Android user base.

To the best of our knowledge, we are the first to find that API updates are more defect prone than other types of changes in client code. Fast-evolving APIs are used more, but the time taken for API adoption is longer. This slow adoption trend may pose various types of risks for client applications such as security vulnerability or poor performance. According to the American Civil Liberties Union (ACLU), "the lag in software updates leaves smartphone users with out-of-date and dangerous systems" [3]. ACLU filed complaints on such spotty Android updates, stating they could potentially harm users by letting hackers steal user data by utilizing security holes. As a part of future work, we would like to understand how the speed of API adoption affects software reliability.

Various stakeholders affect the process of API adoption in the software ecosystem, and further studies are needed to identify factors affecting API adoption. We believe that our findings are a crucial first step and inform future studies on how to promote API adoption and ultimately facilitate the growth of software ecosystems.

Acknowledgment

This work was supported in part by the National Science Foundation under grants CCF-1149391 and CCF-1117902.

References

[1] http://androidheadlines.com/2012/11/2/android/, 2012. [Online; accessed 23-June-2013].

[2] http://stackoverflow.com/questions/3779/, 2012. [Online; accessed 23-June-2013].

[3] Aclu: Android fragmentation creates privacy risk. http://appleinsider.com/articles/13/04/20/aclu-android-fragmentation-creates-privacy-risk, 2012. [Online; accessed 24-April-2013].

[4] Android platform versions. http://developer.android.com/about/dashboards/index.html, 2012. [Online; accessed 8-April-2013].

[5] International Data Corporation Worldwide Quarterly Mobile Phone Tracker. http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37.UWMCM5OR98E, 2012. [Online; accessed 26-March-2013].

[6] The many faces of a little green robot. http://opensignal.com/reports/fragmentation.php, 2012. [Online; accessed 8-April-2013].

[7] S. Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance*, 13(4):263–, Sept. 2001.

[8] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 359, Washington, DC, USA, 1996. IEEE Computer Society.

[9] B. E. Cossette and R. J. Walker. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *FSE '12 Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, New York, NY, USA, 2012. ACM.

[10] D. Dig and R. Johnson. The role of refactorings in api evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[11] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 427–436, Washington, DC, USA, 2007. IEEE Computer Society.

[12] D. Dig, S. Negara, V. Mohindra, and R. Johnson. Refactoring-aware binary adaptation of evolving libraries. In *Proceedings of the 30th International Conference on Software Engineering*, pages 441–450, 2008.

[13] T. Ekman and U. Asklund. Refactoring-aware versioning in eclipse. *Electron. Notes Theor. Comput. Sci.*, 107:57–69, Dec. 2004.

[14] J. Henkel and A. Diwan. Catchup!: Capturing and replaying refactorings to support api evolution. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 274–283, New York, NY, USA, 2005. ACM.

[15] D. Hou and X. Yao. Exploring the intent behind api evolution: A case study. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 131–140, Washington, DC, USA, 2011. IEEE Computer Society.

[16] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of refactorings during software evolution. In *ICSE' 11: Proceedings of the 2011 ACM and IEEE 33rd International Conference on Software Engineering*, 2011.

[17] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.

[18] M. Lungu. Towards reverse engineering software ecosystems. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 428–431, 2008.

[19] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance*, page 120. IEEE Computer Society, 2000.

[20] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 5:1–5:6, New York, NY, USA, 2011. ACM.

[21] R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to api deprecation?: the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 56:1–56:11, New York, NY, USA, 2012. ACM.

[22] I. Ruiz, M. Nagappan, B. Adams, and A. Hassan. Understanding reuse in the android market. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 113–122, 2012.

[23] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, and P. Bringas. On the automatic categorisation of android applications. In *Consumer Communications and Networking Conference (CCNC), 2012 IEEE*, pages 149–153, 2012.

[24] A. Shabtai, Y. Fledel, and Y. Elovici. Automated static code analysis for classifying android applications using machine learning. In *Computational Intelligence and Security (CIS), 2010 International Conference on*, pages 329–333, 2010.

[25] M. D. Syer, B. Adams, Y. Zou, and A. E. Hassan. Exploring the development of micro-apps: A case study on the blackberry and android platforms. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:55–64, 2011.

[26] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 112–118, New York, NY, USA, 2006. ACM.

[27] B. Womack. Google Says 700,000 Applications Available for Android. http://www.businessweek.com/news/2012-10-29/google-says-700-000-applications-available-for-android-devices, 2012. [Online; accessed 1-April-2013].

[28] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported - an eclipse case study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Washington, DC, USA, 2006. IEEE Computer Society.

[29] S. Yau, J. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *Computer Software and Applications Conference, 1978. COMPSAC '78. The IEEE Computer Society's Second International*, pages 60–65, 1978.

[30] J. H. Zar. Significance Testing of the Spearman Rank Correlation Coefficient. *Journal of the American Statistical Association*, 67(339):578–580, 1972.