2015 2nd ACM International Conference on Mobile Software Engineering and Systems

Web API Fragility: How Robust Is Your Mobile Application?

Tiago Espinha
Delft University of Technology
The Netherlands
t.a.espinha@tudelft.nl

Andy Zaidman
Delft University of Technology
The Netherlands
a.e.zaidman@tudelft.nl

Hans-Gerhard Gross
Esslingen University
Germany
Hans-Gerhard.Gross@hs-esslingen.de

***Abstract*—Web APIs provide a systematic and extensible approach for application-to-application interaction. A large number of mobile applications makes use of web APIs to integrate services into apps. Each Web API's evolution pace is determined by their respective developer and mobile application developers are forced to accompany the API providers in their software evolution tasks. In this paper we investigate whether and how mobile application developers deal with the added distress of web APIs evolving. In particular, we studied how robust 43 high profile mobile applications are when dealing with mutated web API responses. Additionally, we interviewed three mobile application developers to better understand their choices and trade-offs regarding web API integration.**

I. Introduction

Modern-day software development is inseparable from the use of Application Programming Interfaces (APIs) [1]. Software developers access APIs as interfaces for code libraries, frameworks or sources of data, to free themselves from low-level programming tasks or speed up development [2]. In contrast to statically linked APIs, a new breed of so-called web service APIs offer a systematic and extensible approach to integrate services into (existing) applications [3], [4], [5]. However, what happens when these web APIs start to evolve? Lehman and Belady emphasize the importance of evolution for software to stay successful [6], [7], and updating software to the latest version of its components, accessed through APIs [8]. In the context of statically linked APIs, Dig and Johnson state that *breaking changes* to interfaces can be numerous [8], and Laitinen says that, unless there is a high return-on-investment, developers will not migrate to a newer version [9].

When integrating with a web API however, developers can no longer afford the inertia that was noted by Laitinen. The web API provider sets the pace for migrating to newer versions (eventually removing older ones altogether) and client developers are forced to migrate. In the statically linked API context, developers could choose to stay with an older version of e.g. libxml, which meets their needs, yet, with web service APIs the provider can at any time unplug a specific version (and functionality), thus forcing an upgrade.

Indeed, while Laitinen claims that client developers will postpone migration to newer versions until there is a high return-on-investment, in previous work [10], [11] we found that some web API providers are eager to push breaking changes and force client developers to migrate to a newer version within a period as short as 4 months.

Through their loose coupling [12] and REST interfaces, web APIs can easily be integrated into applications with a single HTTP request [13]. However, as the integration becomes as simple as exchanging HTTP requests, do client-side developers consider the consequences of ever-evolving web APIs [14]?

We choose to perform our investigation into web APIs in the realm of mobile applications. This was a conscious decision as (1) mobile apps connected through web APIs are an integral part of the mobile computing experience [15] and (2) in the United States of America mobile applications' Internet usage has in 2014 surpassed Internet usage on desktop computers [1]. Being aware of the ever-growing importance of and reliance on web APIs [16], particularly in the mobile computing domain, we wonder how well-prepared some of the most popular Android mobile applications, in some cases used by millions of users, are with regard to a number of factors which include:

- changes and faults in the web API response from the server due to evolution of the web API
- interrupted HTTP requests due to e.g. loss of Internet connectivity
- empty response messages due to server overload

In order to steer our research, our main research question is *"How well-prepared are Android mobile applications with regard to changes in response messages from the web API?"*, which we then divide into the following sub-research questions:

[RQ1] How robust are mobile apps when the web APIs being used return unexpected responses?

[RQ2] Have web API client developers developed resilience against changes in or failure of the web API?.

To address these questions we performed a study on 43 Android mobile applications which make use of at least one web API. In our study we perform fuzz testing [17] through a series of manual mutations aimed at mimicking potential real-world scenarios where the web API changed its behavior either through (communication) failure or software evolution. We then report the different reactions displayed by the mobile applications when dealing with such mutated responses and

[1] http://money.cnn.com/2014/02/28/technology/mobile/mobile-apps-internet/, last visited December 26th, 2014.

978-0-7695-5566-9/15 $31.00 © 2015 IEEE
DOI 10.1109/MobileSoft.2015.9

interview three developers of some of the aforementioned applications as a means to gather deeper insight on whether these developers are aware of the added challenges introduced by web APIs.

The remainder of this paper is structured as follows: in Section II we describe our approach for analyzing web API client robustness, in Section III we describe our experimental setup including how the mobile applications were selected, the added dimensions we analyzed and details regarding the developer interviews, Section IV describes the results of our experiment with mutation analysis as well as the insights from the developer interviews, Section V discusses potential threats to validity of our work, in Section VI we present related work and lastly we present our conclusions and future work in Section VII.

II. Approach

In order to investigate how robust Android mobile applications are when dealing with changing and/or faulty web APIs, we employ a mutation analysis approach (also found in the area of software testing [18]). We apply this approach on web API responses by intercepting such messages before they are received by the Android application. We chose mutation analysis as it allows for the creation of potential mutant web API responses and therefore makes it possible to simulate real-world web API pains. An additional reason for choosing mutation analysis is the fact that we lack metadata on each web API response, and therefore a more targeted approach (e.g. removing only *"non-optional"* fields) is impossible.

In this section, we first introduce our mutation analysis in Section II-A, after which we explain the technical setup that we have used to apply the mutations in Section III-B.

A. Mutation Analysis — Mutant Generation

Our mutation analysis consists of mutating the web API response for a particular web API request sent by a mobile application. Xu et al. [19] set forward four perturbation primitive operators for mutating XML documents: two insertion operators and two deletion operators where the difference is the position in the XML tree where new nodes are added and deleted. One of the addition operators which inserts nodes at the same level as existing nodes is also included in our study. The other addition operator relates to datatype insertion and since none of the studied web API responses contain datatype definitions, it was excluded from our study. The same reasoning was applied to the deletion operators.

Thus, for the purpose of generating web API response mutants, we devise two operators from the aforementioned work: removal of existing nodes and addition of new unrelated nodes (referred to in this paper as *field removal* and *field addition*, respectively). We extend these operators with four other operators: *malforming a response*, *replying with an empty message*, *changing the implicit data type of a field* and *disrupting the data formatting*.

The choice of mutation operators is also supported by the work of Wang et al. [20]. Through a study on the evolution of some of the most popular RESTful APIs, the authors found that all of the proposed mutations (with the exception of the data formatting disruption) are in fact common change types to the APIs' interfaces. The authors also analyzed StackOverflow questions and concluded that questions related to the deletion of parameters (or *field removals*) are in fact one of the three most common questions regarding web API changes.

Field removal mutations consist of removing fields from the web API response. This particular mutation is used as a means to test robustness against *breaking changes*. Whether using a structured approach such as semantic versioning [2], where only major versions are allowed to bear breaking changes, or using a more lenient approach, when dealing with a web API it is possible that some fields are removed from web API responses. Examples of breaking changes caused by the removal of fields can be found in our previous work [10], in particular regarding the Facebook web API [3]. Such breaking changes, as defined by Dig and Johnson [8] (in the context of statically linked APIs), remind us that when fields are moved, changed, renamed or replaced, a field is always inevitably removed (e.g. a rename is a removal plus an addition with a different name). Also Li et al. [21] show that indeed more providers rename parameters which also results in breaking changes.

While applying this mutation, removing fields in different parts of the web API response has the potential to result in different behaviors. For this reason we perform a step by step removal where after each step, the mobile application is tested against the resulting mutated web API response for that step. Our step by step removal is performed using the following removal guidelines to ensure that each child node of each node type is removed at least once. Since no data type definitions were encountered, we assume a node is of the same type as a different node when they share the same children structure.

Removal Guidelines.

Rule #1 — A node of each type is removed in its own step once.

Rule #2 — If more nodes of the same type (as an already removed node) exist at the same level, they are left to be removed upon the removal of their parent node.

Rule #3 — After each bottom-most level has been emptied (or *only* contains nodes, the type of which, one node has already been removed), we move up a level.

Rule #4 — In each level, if nodes of different types exist but neither have children (or have children of *repeated* types), all these nodes are removed in one step.

Example. As an example, we analyze the node tree of Figure 1 where each node represents a web API response field. In this node tree, the nodes are named after their type.

Step 1 — In the bottommost level (i.e. 1st level) two nodes exist of the same type. According to Rule #1 we remove **node a** and due to Rule #2, **node a'** is left for later removal.

Step 2 — Due to Rule #3 we move up to the 2nd level. At this stage, **node b** is removed and **node b'** is *ignored*.

[2]Semantic Versioning — http://semver.org/

[3]Facebook Completed Changes — http://bit.ly/fb-completedchanges

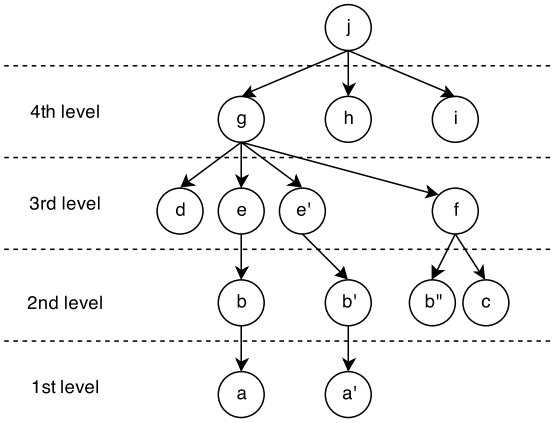

Fig. 1. Field removal mutation applied on a sample node tree

While **node b"** is *also* of the same type as **node b**, its parent node is of a different type (**node f**, which has one more child). Therefore, **node b"** and **node c** are removed in separate mutations.

Step 3 — Having cleared the 2nd level, we move to the 3rd level and all the nodes are child nodes of **node g**. Therefore, according to Rule #4, **nodes d, e, e' and f** are removed in one mutation.

Step 4 — Again due to Rule #3, we move up to the 4th level and apply Rule #4 to remove the remaining nodes (**nodes g, h and i**) in one mutation. After this step, only the root node is left, which is never removed.

Of note is the fact that despite removing fields, special care was taken to keep the web API response semantically valid. This was done through the usage of online validators for both JSON [4] and XML [5] documents.

Malformed responses can happen for a number of reasons. While not necessarily a direct result of software evolution, such responses happen e.g. if the data encoder of the web API fails to properly sanitize strings. Such failure could then lead to floating reserved characters which in turn break the document's data format. This can happen while, for example, encoding a JSON string which contains double quotes and these are not properly escaped (i.e. "foo":"b"ar" as opposed to the valid "foo":"b\"ar"). In order to mimic these failures, our malformed response mutation consists of mutating a random node of the web API response as to make it malformed in its respective data format. In XML we mutate the response by breaking an XML tag (e.g. '<data>' becomes '<data') and in JSON this is achieved by leaving a dangling double-quote in a JSON string (e.g. "foo":"bar" becomes "foo":"bar).

Empty responses can be a symptom of different types of ailments on the web server. For instance, should the web server be at the edge of its maximum capacity, some requests may receive an empty response. Similarly, if the connection is terminated due to communication issues, it could also lead to empty responses being returned to the mobile application. Our empty response mutation consists simply of replacing the response with an empty-bodied HTTP response.

Changing data types mutations consist of two changes: selecting at random a numeric field and replacing it with a string as well as performing the reverse operation on one other, randomly selected, string field. As all the mobile applications under study run on Java (by force of the Android platform) and since Java is a statically typed language, if special care is not taken when parsing the web API response, type mismatches could occur. Also as a part of software evolution, web APIs may at some point change the (implicit) data types of certain fields. For instance, while the price field can be a string which includes the currency symbol, at a later stage the price field can become a purely numerical field.

An important remark is that no metadata is available for any of the web API responses under study. Therefore, this particular mutation is done by changing a field which visually appears to be of a type into the other type.

III. Experimental Setup

For our experiment, we first required a body of mobile applications to be analyzed. How these applications were chosen is described in detail in Section III-A. We then introduce mutations in web API responses through the use of a transparent web proxy, described in Subsection III-B. These mutations, previously described in Subsection II-A, are a means to probe the robustness of the mobile applications by emulating breaking changes and faults of the web API.

We also expand this quantitative view with a more qualitative approach. To do so we interviewed 3 developers, each from a different application under study. More details regarding these interviews are also presented.

A. Application Selection

With our study we aim to understand what is the current state of web API integration issues. We do so by analyzing some of the most popular Android applications. All the applications under study were required to meet two criteria.

The first criterion is that each application must use at least one web API. Important to note is that our definition of web API excludes mobile applications which simply load HTML or RSS feeds. These exclusions are due to the fact that with an HTML response, no processing is required on the mobile application. With this we exclude also any mobile application which may use screen-scraping techniques [22] as screen-scraping is performed by extracting data from user interface elements which are not necessarily designed with the same backwards compatibility concerns of an API. Similarly, RSS feeds have a fixed structure which means they are not susceptible to software evolution changes which make them not applicable for our study.

The second criterion is that the web API communication happens over an insecure channel such as HTTP as opposed to HTTPS. Indeed, this is particularly important as an encrypted protocol such as HTTPS does not allow for changes to be made to the content of its messages (without a security certificate).

[4]JSON Editor Online — http://www.jsoneditoronline.org
[5]XML Viewer — http://codebeautify.org/xmlviewer/

For this study we picked candidate applications from the top 100 free applications available in the Google Play Store of the Netherlands, United Kingdom, United States of America, Canada, Australia, Belgium, Brazil, Spain, Germany and France. Many of the apps in the different countries overlap and in the end we installed 198 applications. We had to exclude 68 of these because they use HTTPS, while a number of other applications use proprietary binary data formats for the communication. Similarly, other applications which use compressed files as a means to transfer the web API responses (e.g. ZIP files) were not mutable using our approach as it would require the response to be decompressed and recompressed on the fly. Ultimately, our study corpus is composed of 43 applications (see Table I [6]).

[6]The Buienalarm and Eurosport mobile applications make use of two distinct web APIs and therefore appear twice on the list.

TABLE I
LIST OF MOBILE APPLICATIONS STUDIED

Application	Version Number	Crashes	Versioning	Caching
NS.nl	3.2.1	√	√	√
yr.no	3.0.1	√		
Skyscanner	2.0.9		√	
Buienalarm (own API)	2.3			
Buienalarm (OpenWeatherMap)	2.3			
BBC News	2.5.2 WW	√		
Daily Mail Online	3.2		√	
WeatherBug	3.5.97		√	
The Weather Channel	5.0.3	√		
Reddit is fun	3.3.12	√		
Ozsale	3.3.12		√	
tramTracker	1.3	√		
NRL - League Live	4.5.7			
The Masters Gold	4.1			
mobile.de	4.2.0	√		
Wetter App	2.3.3			
MeinProspekt	7.19		√	
TV Movie	1.3.1	√	√	
Wetter.com	1.4.9.4	√		
McDonald's Deutschland	1.4.0.1			√
H&M	2.6.1			
eltiempo.es	1.1.2	√		
Liga de Futbol Professional	5.2.12	√		
TecMundo	1.6.1d	√	√	
Trivago	1.9.4		√	
BeSoccer	3.0.3	√		
La Chaine Meteo	1.1.3		√	√
Resultats Foot en Direct	2.8			
RATP	3.0.10			
ViaMichelin	3.5.0		√	
Le Figaro	3.5.1		√	√
Le Parisien	3.3.2			√
Eurosport (XML)	3.7.6			
Eurosport (JSON)	3.7.6			√
Tele Loisirs	4.4.1			
NU.nl (stocks)	5.4.1		√	√
Just Eat	1.4.1.63			
Couverts	2.8.3			
Trulia	5.7.2		√	√
24Kitchen	2.2			
NL Treinen	2.0.10			√
Huizen	1.71		√	√
Kieskeurig	0.9.9.2		√	
Jumbo FoodMarket	1.1		√	√
Pull&Bear	1.2.3		√	
	Total	13	18	11

Of note is the fact that, despite not having been a criterion for selection of the applications, all of the analyzed applications make use of either XML or JSON as a data format for the web API requests. This is particularly relevant as a data format like XML requires an additional XSD schema document to enforce data types. For this to happen, the XML document would require references to the XSD [7] document which can be used to validate it. Such cases have not been encountered in this study which means no XML documents were being (explicitly) validated against an XSD schema. During our mutation analysis, changing data types was still attempted on XML documents (i.e. changing a field with a numeric value to a string) even though none of the fields were specifically numeric as it happens with JSON where numeric fields can be identified by the absence of quotation marks.

B. Applying the Mutations

In order to mutate the responses from each mobile application's respective web API, we start by installing the chosen applications for our study on a Google Nexus 7 tablet (running Android KitKat 4.4.2) and configure the tablet to redirect all the network traffic through a transparent proxy (Charles Web Proxy) setup in a separate machine on the same network.

For each mobile application studied, before starting the mutation analysis, we follow a series of manual steps:

1) While using the transparent proxy, we identify a repeatable action (e.g., the push of a button) which causes a request and response interaction with the web API.
2) We then collect a standard response (we made all the standard responses, many of which several hundreds of lines long, available online [23]) for that particular web API request made by the application under study and configure the transparent proxy to replace the response of all other similar requests (for the purposes of the Charles web proxy, a similar request means a request sent to the same endpoint) with a customized response.
3) The customized response is in fact the original web API response although slightly modified. The first "modification" that we try is actually a message identical to the original one, in order to ensure that messages that we intercept and change are actually loaded by the mobile app. All subsequent customised response messages do in fact undergo the mutation analysis.

After ensuring the customized responses are replacing the standard response, the original web API response is then further disturbed with a number of different types of mutations (explained in detail in section II-A). For each mutation we observe how the Android application reacts to such changes. Our observations are then categorized into different types of behaviors (e.g. crashing or indefinitely loading without a timeout) and turned into a report (a sample is available online [8]) which is two-fold in its content: it starts with a statistical overview displaying how many applications behave

[7]XSD Schemas — http://www.w3.org/TR/xmlschema11-1/
[8]Sample status report — http://bit.ly/report-web-api

in each of the identified categories, and culminates with a report specific to that particular application. In particular, we provide data on how many applications crash due to a mutation, how many use versioning and the divide between JSON and XML implementations. Also for each type of mutation we provide statistics on the different observed behaviors. We also present statistics on how many applications use caching and complement the report with application-specific findings (e.g. some applications still load malformed data). It is also these outlier findings of behaviors that are not common which we aim at clarifying with the developers through the interviews.

C. Caching and Versioning

After beginning our experimental study, we found that for some applications, after the *"sample web API response"* was collected to be used as basis for the mutations, the mutated data (even with a string for string mutation) would not be loaded. This hinted at the usage of caching on some of the mobile applications. As caching is of particular interest for mobile applications where the Internet connectivity may sometimes experience slow bandwidth and where the web API may not respond due to high peaks of server load, we collected data on whether each mobile application uses caching and whenever possible, how the caching is used.

For what concerns versioning, in previous work [10] we found that some high-profile web APIs (e.g. Twitter, Google Maps, Netflix) make use of some form of versioning. Still, we also found a major web API provider (Facebook) which does not make use of any form of versioning in their web API. Having studied these two different approaches and how client developers perceive each of the aforementioned web APIs, we also collect data on which web APIs are versioned.

D. Developer Interviews

While the empirical study described above provides interesting insight on how a large body of mobile applications react when web APIs experience different failures and changes, it does not provide an explanation as to the choices of their respective developers. As an attempt to shed some light on the developer perspective of developing and testing an application which integrates with a web API, we aimed at interviewing the developers of some of the mobile applications under study. These interviews took the format of a semi-structured interview [24] using the questions in Table II as a basis to stimulate the exploratory discussion.

While the questions listed in Table II are primarily targeted at client-side developers, some of the questions are related specifically to developers that also have knowledge of the web API development side. In particular, **Q6** gauges as to whether the server and client-side are developed by the same team. The answer to this question can in turn determine the answer to questions like **Q1** and **Q3** (because these decisions lie primarily with the web API developers and not the client developers), **Q4** and **Q9**.

We selected 14 applications which stood out either due to a special versioning mechanism, because they crashed or because of some particular behavior that other mobile applications did not demonstrate. We sent the reports from Section III-B to the respective developers of these 14 apps, along with an invitation to participate in an interview.

Ultimately, only 3 developers responded to our request for an interview: the mobile software architect for *OZsale*, the product manager for the *Trivago* mobile application and lastly, the sole developer of the *NS.nl* Android application. The interviews lasted 15 minutes on average. While the insights thus obtained are not enough to build solid conclusions on, they do provide us with valuable anecdotal evidence.

The *status report* we compiled and sent to the developers contains statistical information on all the applications under study. We provide data on how many applications crash due to a mutation, how many use versioning and the divide between JSON and XML implementations. Also for each type of mutation we provide statistics on the different observed behaviors. We also present statistics on how many applications use caching and complement the report with application-specific findings (e.g. some applications still load malformed data). It is also these outlier findings of behaviors that are not common which we aim at clarifying with the developers through the interviews.

IV. Experimental Results

We present our findings regarding each observed behavior in the Android applications under study upon applying the different mutations to the web API response. Specifically, we report on four of the six initially proposed mutations as the field addition and data formatting mutations did not elicit any unexpected behavior. We provide an analysis of the different behaviors displayed by the mobile applications and whenever relevant, provide anecdotal examples.

We also present our findings on the different types of data caching and web API versioning encountered as well as developer input on some of the choices used in web API integration. Of note is that the results are valid for the respective versions studied (see Table I), as each of the mobile applications and their respective web API may change, so may the mobile applications' reaction to web API mutations.

TABLE II
Questions Asked During the Developer Interviews

Q1	What was the design decision behind choosing HTTP over HTTPS?
Q2	Why are no caching mechanisms used?
Q3	Is versioning not used for a particular reason?
Q4	Is the web API used by other (third-party and or mobile) applications?
Q5	Is the mobile application native to Android or generated with a mobile development framework?
Q6	Is the mobile application developed by the same team as the web API?
Q6.1	In particular when it is not developed by the same team, how does the mobile application team learn about the web API changes?
Q7	How frequently are the web API and mobile application updated?
Q8	Have there been problems in the past with breaking changes causing the mobile application to break?
Q9	Are there automated tests in either the mobile application or web API?

A. Application Behavior

When each of the different types of mutants was applied to the web API responses, we observed four distinct behaviors from the mobile applications:

1) Force close — The *force close* is Android terminology for applications which crash by throwing uncaught Java runtime exceptions. Such exceptions are then caught by the Android platform and the application is *force closed*.
2) Error message — Showing an *error message* is a graceful way of letting the end-user know that something did not go as expected and what his or her course of action should be (e.g. try again or check the Internet connectivity). Some applications show an error message whenever a number of disturbances afflict the connection to the web API server. Whenever these error messages are shown, the robustness criteria is met as the applications do not crash and it is therefore a preferable behavior to a force close. However, in the highly dynamic domain of web APIs where new versions are released regularly and often without the client developers' knowledge, we expected some applications to provide recommendations in the error messages as to what the end-user should do. This was never the case and in fact, the end-user is at times misinformed about the nature of the problem.
3) No indication — By opposition to showing an error message, some applications silently deal with the mutated web API response. In some cases the data is partially loaded, in other cases a clue is provided that the loading has stopped, but no information is given to the end-user as to what has happened. By not showing any reaction to the end-user's input, the mobile application may induce confusion. When the application simply does not react even though the request has been made to the web API and the mutated web API response has been received, it is impossible for the end-user to know whether the data is still being loaded (as in these cases there was also no visual indication of loading) or whether he or she should try again to refresh the data.
4) No timeout — timing out is an important part of reporting a failure. At times however, applications remain indefinitely loading. It is then up to the end-user to decide when to stop waiting and close the application. Such behavior indicates that whatever exceptions may have been thrown are being handled (since the application did not crash) but are potentially being muffled. Ultimately, the application never closes the loading screen and fails to provide the end-user with insight as to what happened.

We now present each of the mutant types and how each of the behaviors have been observed per mutant.

B. Behaviors Per Mutation Type

1) Field Removal: As explained previously in subsection II-A we expected field removal mutations to be the mutations which more faithfully represent a web API evolution scenario. This particular mutation was applied in a particularly invasive way (i.e. all the child nodes of each type have been removed at least once) which coupled with its disruptive nature (i.e. the message is valid but is missing data) lead us to expect such mutations to be the most challenging in testing the robustness of the mobile applications.

Force Close. Our results show that 13 applications out of the 43 under study (approximately 30%) when faced with field removal mutations crash with a *force close*. Using our approach also allowed us to narrow down on exactly which fields were causing the applications to crash. In fact, 12 out of the 13 crashing applications crash with *one single field* being removed (the particular field is found following the steps in Section II-A). The remaining application required two fields to be removed in order to crash. While we have no metadata for any of the web API responses, all the 13 crashing applications allowed for *some* fields to be removed without crashing.

This result also confirmed our initial expectation. As can be seen in Table III the field removal mutation is the one which resulted in the most applications crashing.

Web API ownership may also play a role in hardening a client application against web API changes. For example, the *NS.nl* web API is also used by a third party application (*NL Treinen 2 - NL*) which does not crash with field removal mutations (as opposed to the *NS.nl* application). When asked about this, the interviewed developer for the *NS.nl* application claimed he was in control of the web API and therefore knew when the web API would change. Unfortunately the developer for *NL Treinen 2 - NL* did not react to our interview request and we cannot verify our hypothesis.

Error Message. When certain fields were missing from the web API response, there were 4 out of the 43 applications which did show error messages. In some cases the error message is generic (e.g. *"connectivity issue"*) and not specially tailored to inform the user on how to proceed.

No Indication. More troubling than showing an incomplete error message is the behavior displayed by 39 out of the 43 mobile applications under study (n.b. the 39 applications which show no indication include the 13 which force close). These applications show no indication of completion or failure whenever one or more fields of the response are removed.

2) Malformed Response: Mutations of the type malformed response resulted in the same three different behaviors as the field removal mutation, with an added behavior where applications failed to timeout.

Force Close. Having one single application crashing upon receiving a malformed response (namely *Wetter.com*) is an indicator that the majority of the applications are accounting

TABLE III
APPLICATION BEHAVIOR VERSUS MUTATION

Mutation \ Behavior	Force Close	Error Message (vs Silent Fail)	Timeout (vs Indefinitely Loading)	No Indication
Field Removal	13/43	4/43	-	39/43
Malformed Response	1/43	12/43	35/43	31/43
Empty Response	1/43	9/43	35/43	34/43
Changing Data Type	3/43	3/43	-	40/43

for the scenario where the document parser fails due to a faulty document. One other application (*Le Parisien*) would become unusable upon facing a malformed response but rather than force closing, it would report to the user that it cannot continue and then gracefully close itself.

Error Message. The *NS.nl* application, when facing a malformed response makes use of the Android native dialog mechanism to show a message informing the end-user that it *"cannot retrieve data from server"*. While certainly better than crashing or remaining silent about the failure, it is also an example of a generic message which does not offer an indication of how the end-user should proceed.

No Indication. While malformed responses are an unmistakable case of failure somewhere along the connection with the web API, it is then surprising that 31 applications (72%) give no indication of the unrecoverable error or of what is the right course of action for the end-user.

Also in this category, we found two different types of applications. In some cases, such as the *tramTracker*, the application still attempts to load whatever data is available in the damaged response (whilst giving no indication that it was damaged). In contrast, applications such as the *Resultats Foot en Direct* load directly onto a screen where if the response is malformed, nothing is shown. In such a case, the end-user may be endlessly waiting, expecting the data to be loaded while the application has in fact silently stopped loading.

No Timeout. Applications which indefinitely stay loading and never timeout was the most common occurrence when dealing with malformed responses. Indeed, 8 applications (19%) never time out and are left stuck in a loading screen.

3) Empty Response: When applying the empty response mutation, all four behaviors were observed. While other mutations consist of turning the web API responses into crash-inducing mutants, we expected an empty response to be a fairly trivial occurrence without serious repercussions.

Force Close. Indeed, while only one application demonstrated this behavior, it stands to reason that the source code of *TecMundo* does not account for empty web API responses. When presented with an empty web API response, this application would immediately force close. All the other 42 applications did not crash when faced with such an empty response.

Error Message. When dealing with an empty response, 9 applications of those under study did show an error message. Of special note are 4 of these applications (*mobile.de*, *Resultados Futbol*, *Couverts* and *Trulia*) which show a message claiming that no results were found and that the end-user should change the search criteria. In fact, all these applications always return a boilerplate JSON or XML result *even* in the event no results had been found. This may then indicate that the application did not recognize the empty response as a fault. The remaining 5 applications showed generic *"network error"* messages, with special attention to the *yr.no* application. This was the only application which actually reported an *"empty response"*.

No Indication. In contrast with the low number of applications which display an error message, more than two thirds (34 out of 43) of the applications under study provide no indication that an empty response has been received from the web API. More specifically, these mobile applications would stop loading and never present the user with a message describing why the loading had stopped and why no data had been loaded.

No Timeout. In part overlapping with the applications which provide no indication of receiving an empty response, 8 out of the 43 applications remain indefinitely loading and never time out. While it is not possible to verify the reason for this behavior in the applications' source code due to their closed source nature, it is likely the affected applications always expect a reply with content. When the content is not present, the applications hang until there is content.

In fact, RESTful web APIs may indeed at times reply with an empty message, for example with HTTP status codes 304 (Not Modified) or 204 (No Content) [9]. Although it was not the case for any of the aforementioned 8 applications, in our study we experienced web APIs which replied with an empty HTTP message having a status code of 301 (moved permanently), indicating that a request should be made to a different URL.

4) Changing Data Type: Due to the lack of metadata on all the web API responses, it is impossible to know with certainty whenever a field is of the numeric or string type. Nonetheless, with the exception of the timeout issues, our mutated web API responses were able to cause all of the other aforementioned behaviors (force close, error message and no indication).

Force Close. Through the use of the changing data type mutation, 3 applications crashed. While we were not able to investigate the exception being thrown, it is possibly related to the parsing of the message and to a type mismatch between Java's statically typed variables and the field values being parsed into the wrong types.

Error Message. The changing data type mutation resulted in only 3 applications actually showing an error message. Out of the 3 applications, none provided an error message which offered an explanation or solution for the problem.

No Indication. As with the other faults, some of the mobile applications silently fail without informing the end-user about the fault. In fact, 40 out of the 43 applications did not report an error even when the data would not load (in which case we were certain to have disrupted the loading of the data).

C. Data Caching

One way for client developers to build in resilience against unexpected changes in the web API response (e.g. due to communication errors) is to make use of a local data cache. In our investigation we encountered a number of mobile apps that would not initially attempt to load the mutated data. This was due to the fact that because the data had just successfully been loaded (from our first execution, probing for a testable action), the data would be stored in cache.

This led us to investigate how many of the mobile applications under study were making use of caching. The results of this investigation can be observed in Table I and show that the majority (32 out of 43 applications) do not use caching.

[9]REST Patterns, HTTP Status Codes — http://bit.ly/restpatterns

D. Versioning

In previous work [10] we highlighted the importance of versions in the web API context. Especially when the client developers have no control over when changes happen to the web API behavior, versioning that behavior allows the client developers to know what behavior to expect from a particular web API. Versioning, either in the URL (e.g. www.weather.com/v1/report) or through variations of semantic versioning (as demonstrated by *OZSale*), allows client developers to know *when* to expect changes. Indeed, in the case of *OZsale*, the web API currently at version 3.4 is guaranteed to not introduce breaking changes in all the minor versions of the 3.X release.

It is then surprising how such a high percentage of mobile applications make use of the web API without any form of versioning (58%). When a non-versioned web API introduces changes, all the clients which have not yet migrated to the latest version will be interacting with a changed web API, which may not be compatible. Evidence of a scenario where this would potentially happen would it not be for versioning comes from one of the interviewed developers. The developer interviewed in the context of the *OZsale* application claimed that indeed, some end-users do not update their mobile applications and that 5% of their user-base (of 100,000∼500,000 users according to the Google Play Store) was still using their mobile application's very first version.

E. Developer Interviews

We conclude our study with the findings gathered from interviewing the three participants in our study. In the paragraphs below we refer to the questions shown in Table II.

Insecure HTTP. Referring to question **Q1** regarding the design decision of using HTTP over HTTPS, an intriguing finding of our study is how such a large number of mobile applications (indeed, all the 43 under study) still make use of insecure HTTP. Data sent over HTTP allows for the data to be both eavesdropped upon and, indeed, tampered with in the same fashion as is performed in this study. When confronted with this question, one of the developers claimed he did not in fact know why their web API was using HTTP because the web API is developed by a different team. According to the developer their mobile application does make use of HTTPS for login and payment interactions.

Caching. While all the interviewed developers perceive caching (question **Q2**) as a useful mechanism to reduce network usage, specifically the developer of the *NS.nl* application raised a concern about the necessity for *"fresh data"*. Indeed, this application provides information on the Dutch train departure and arrival times which are at times susceptible to delays. It is thus crucial to always display the latest data.

Another interviewee (the mobile software architect for *OZSale*) justified the lack of caching as it being a lower priority requirement. While such a feature is already present in the iOS version of the mobile application, at the time the Android version started being developed *"the libraries available for caching in the Android platform were not yet mature enough"*. The iOS application goes a step further and makes all of the data available for offline browsing.

Also the developer responsible for the web API at *Trivago* stated that caching would in fact stay in the way of the mobile application's performance for their specific case. Caching would require the mobile application to keep track of which data it has available and only request the delta between what it already has and the results it still needs to fetch. To do so with caching and without state would make for chatty communications. The mobile application would have, with every request, to report what it already has in cache and what it requires. *Trivago* contains a more pragmatic approach where sessions (i.e. stateful exchanges) are used which allows the cache to be on the server-side and thus lower the chattiness which is desirable for both performance and data usage.

Versioning. Two of the three interviewed developers (for the *Trivago* and *OZsale* mobile applications) have versioning mechanisms implemented in their respective web APIs.

For instance, the *Trivago* web API makes use of HATEOAS [10] (Hypermedia as the Engine of Application State) versioning approach. The HATEOAS approach makes use of HTTP headers (Accept-Type and Content-Type) as a way to handle versioning and description of the data since it stands central to being *the* way a RESTful web API should be versioned. When asked about why this particular versioning mechanism was used, the answer was that even though the Trivago web API is still in its first version, HATEOAS was chosen as a way to future-proof the evolution of the web API.

The developer of the *OZsale* application also stressed the importance of their versioning system. While the data itself is not versioned (as it happens with HATEOAS), a version number must be used in the URL to inform the server of which version of the web API the mobile application requires.

An interesting divide between the two aforementioned developers is how old versions of the web API are handled. The Trivago software architect underlined that they try to avoid maintaining different versions in parallel due to costs, while the *OZsale* developer claimed that the different versions were a core part of their different platforms: while the website was running on the latest version of the web API, the different mobile platforms were lagging at least 5 minor versions behind (all of which were still available and fully functional). A reason for this was the delay between submitting a new version of the mobile application to the respective application store and the application actually being available (e.g. the developer claimed a 1 week delay in the iOS App Store).

The developer of the *NS.nl* mobile application claims that over the course of four years of development, no breaking changes have been applied to the web API, thus making a versioning mechanism unnecessary.

Evolution & Communication fragility. Another interesting finding is anecdotal evidence of communication issues between the different teams involved in the development process. The developer of the *NS.nl* application is also the

[10] Versioning REST Services — http://bit.ly/versioningrestservices

developer of the web API and therefore reported no such pains. However, the two other interviewed developers relied on separate teams for the web API development. Indeed, one of the developers claimed that at least twice in their project, changes were pushed to the web API which inadvertently broke backwards compatibility. The result was having a mobile application which was crashing. This anecdote raises an issue which is also supported by our analysis of the 43 mobile applications: not all mobile applications are built with the consideration that the web API can change *at any time*. This was especially relevant as in this very same project, changes were being pushed daily with breaking changes taking place every two months highlighting the need for excellent communication between the mobile teams and the web API team should these teams not be one and the same (questions **Q4**, **Q6** and **Q6.1**).

Integration Testing. While using a static library it is possible to test it and expect it to behave the same. However, when using web APIs where the behavior can change due to a simple patch which fixes what was buggy (but expected) behavior can cause the mobile application to suddenly misbehave. This highlights the importance of both positive and negative testing, that is testing both scenarios which are part of the use cases as well as unexpected but potential scenarios.

Our empirical data suggests that Çalıklı and Bener's [25] observation on confirmation bias regarding testing may indeed affect some of the studied applications. Indeed, while some of the applications may have automated tests (which we cannot confirm due to their closed source nature), they may be positive tests which *"make their program work rather than breaking the code"* as would be the case with negative tests. In our interviews, we questioned the participants on whether their application makes use of any kind of testing (**Q9**). Our results show that for some of these applications a simple mutation such as malforming the web API response caused a crash. Considering the *OZsale* application as an example, the interviewed mobile software architect claimed they do perform automated testing for some bad scenarios which may potentially happen, this very same application would remain loading indefinitely when faced with a malformed response.

V. Threats to Validity

External validity. Our study which includes 43 applications, is composed solely of Android applications. Other mobile platforms which make use of web APIs such as iOS or Windows Phone should also be explored. Perhaps in some platforms it is more or less difficult to cause the whole application to crash. As such, in future work we will perform a similar study on both the iOS and Windows Phone platforms.

Similarly, our study can only be applied to mobile applications which make use of the insecure HTTP protocol. This both limits the number of mobile applications which can be used. Mobile developers who intentionally chose to use HTTPS over HTTP are perhaps more conscious regarding the differences which make *web* APIs different from the non-web counterparts. Indeed, without modifying the Android platform itself, nothing can be done to mitigate this threat.

Internal validity. While our study intercepts and mutates web API responses and analyzes mobile applications' reactions to these mutations, we did not consider whether these mobile applications send failure data back to the respective software developers for further analysis. Such data, should it exist, may compliment and aid the debugging task.

Another threat to validity stems from potentially long timeouts (e.g., several minutes) when reacting to mutated web API responses. In such case an application could potentially lead to a misclassified application. While a threat, unnecessarily long timeouts would also potentially hinder the usage of such applications for end-users.

Reliability validity. Our mutation analysis approach requires human intervention when capturing a standard web API response and replacing the response with its mutated counterpart. This raises a possible reliability threat. We mitigate it by starting with a slightly mutated response (e.g., changed string) whilst maintaining its validity as a means to ensure that the mutated response is indeed being loaded.

VI. Related Work

Li et al. [21] highlight the evolution challenges of web APIs over statically linked APIs and provide a set of potential changes which web APIs may implement. They analyze what are common changes applied to web APIs and propose the creation of a tool for automated client migration.

McDonnell et al. study API stability and adoption in the Android ecosystem and have found that, despite the added benefits of newer versions of APIs, developers tend to be slow in adopting the newer versions [26], thus further highlighting the awareness required when web API changes are inevitable.

An interesting non-peer reviewed work in this field is a survey [27] conducted on the pains of web API integration which presents many complaints from web API client developers.

Daigneau focuses on the brittleness of web APIs and proposes to refrain from creating signatures with long parameter lists [28]. Daigneau further states that long parameter lists *"[...] signal the underlying framework to impose a strict ordering of parameters which, in turn, increases client-service coupling and makes it more difficult to evolve the client and service at different rates."*

VII. Conclusion

In this paper we perform a study on the impact that changes to web API behavior can have on mobile applications. Our contributions are:

- An approach using mutation analysis for simulating unexpected responses from web APIs.
- A study on how 43 high profile mobile applications react to a set of predefined mutations in web API responses.
- Insight on caching and versioning approaches of some of the web APIs under study.
- An interview with three developers of some of the studied mobile applications.

Referring back to our research questions proposed in the introduction, we set out to find how robust mobile applications are when facing unexpected responses from web APIs.

In order to address **[RQ1]** which asks *"how robust are mobile apps when the web APIs being used return unexpected responses?"*, we use mutation analysis. Mutation analysis presents a structured approach to simulate web APIs afflicted either by failure or by changes caused by software evolution. Our results present a mixed answer to this question. Indeed, most of the mobile applications studied are fairly robust to mutations in the web API response as seen by only 30% of the applications studied crashing through the field removal mutation. Nonetheless, all but one of the applications can be crashed through the removal of one single field which presents a serious concern for some web API client developers. Less serious but also worrying is how for all the mutations, more than half of the applications silently hide the faulty web API response. This behavior should be made more informative and user-friendly, which can be achieved through better understanding potential changes to web APIs.

Also **[RQ2]** which asks *"have web API client developers developed resilience against changes in or failure of the web API?"* is answered with mixed results. Some of the applications studied make use of state of the art approaches (e.g. the HATEOAS versioning) to ensure a smooth evolution of their web API client, where others do not use versioning altogether (which as reported in previous work [10] may cause long-term pains) and allow the application to crash. The need for this resilience exists also outside of the source code. One of the interviewed developers raised concerns with inter-team communication, highlighting the need for clear and concise documentation from web API providers to client developers.

Our main research question asks *"how well-prepared are Android mobile applications with regard to changes in response messages from the web API"*. We conclude that while the majority of the studied applications are capable of dealing with such changes without major issues, some applications still use web APIs as if their behavior can be expected to never change, which as we have seen does not always happen. Rather than trying to generalize the results for all the web API clients, our goal is rather to raise awareness to the fact that amongst some of the most popular Android applications, a fair share still allow their web APIs to significantly affect their behavior.

Future work. We aim to extend our investigation to paid mobile applications and other platforms (e.g. iOS) as we want to understand whether the underlying platform provides more or less support for web API integration.

Another aspect worth investigating is whether the owner/creator of the web API influences client developers to use validity checks to the web API response.

REFERENCES

[1] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE CS, 2012, pp. 378–387.

[2] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 2008, pp. 481–490.

[3] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI," *Internet Computing*, vol. 6, no. 2, pp. 86–93, 2002.

[4] S. Vinoski, "Restful web services development checklist," *IEEE Internet Computing*, vol. 12, no. 6, pp. 96–95, 2008.

[5] B. Srivastava, "Composing web apis: State of the art and mobile implications (tutorial)," in *Proc. Int'l Conf. on Mobile Software Engineering and Systems (MOBILESoft)*. ACM, 2014, pp. 3–4.

[6] M. M. Lehman and L. A. Belady, *Program Evolution: Processes of Software Change*. Academic Press, 1985.

[7] A. Zaidman, M. Pinzger, and A. van Deursen, "Software evolution," in *Encyclopedia of Software Engineering*. Taylor & Francis, 2010, pp. 1127–1137.

[8] D. Dig and R. E. Johnson, "How do APIs evolve? A story of refactoring," *Journal of Software Maintenance*, vol. 18, no. 2, pp. 83–107, 2006.

[9] M. Laitinen, "Object-oriented application frameworks: Problems and perspectives," M. Fayad, D. Schmidt, and R. Johnson, Eds. Wiley, 1999, ch. Framework maintenance: Vendor viewpoint, p. 9.

[10] T. Espinha, A. Zaidman, and H.-G. Gross, "Web API growing pains: Loosely coupled yet strongly tied," *J. Systems and Software*, vol. 100, p. 27–43, 2015.

[11] ——, "Web API growing pains: Stories from client developers and their code," in *Proc. Conference Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 84–93.

[12] C. Pautasso and E. Wilde, "Why is the web loosely coupled? a multi-faceted metric for service design," in *Proc. Int'l World Wide Web Conf. (IW3C2)*. ACM, 2009, pp. 911–920.

[13] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. "big" web services: Making the right architectural decision," in *Proc. Int'l Conf. on World Wide Web (WWW)*. ACM, 2008, pp. 805–814.

[14] T. W. Knych and A. Baliga, "Android application development and testability," in *Proc. Int'l Conf. on Mobile Software Engineering and Systems (MOBILESoft)*. ACM, 2014, pp. 37–40.

[15] J. H. Christensen, "Using RESTful web-services and cloud computing to create next generation mobile applications," in *Proc. Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA-companion)*. ACM, 2009, pp. 627–634.

[16] M. Maleshkova, C. Pedrinaci, and J. Domingue, "Investigating web APIs on the world wide web," in *Proceedings of the European Conference on Web Services (ECOWS)*. IEEE, 2010, pp. 107–114.

[17] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990. [Online]. Available: http://doi.acm.org/10.1145/96267.96279

[18] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[19] W. Xu, J. Offutt, and J. Luo, "Testing web services by XML perturbation," in *Proc. Int'l Symp. Software Reliability Engineering (ISSRE)*. IEEE CS, 2005, pp. 10 pp.–266.

[20] S. Wang, I. Keivanloo, and Y. Zou, "How do developers react to RESTful API evolution?" in *Service-Oriented Computing*, ser. LNCS. Springer, 2014, vol. 8831, pp. 245–259.

[21] J. Li, Y. Xiong, X. Liu, and L. Zhang, "How does web service API evolution affect clients?" in *Int'l Conf. on Web Services (ICWS)*. IEEE, 2013, pp. 300–307.

[22] J. Martin, A. Arsanjani, P. Tarr, and B. Hailpern, "Web services: Promises and compromises," *Queue*, vol. 1, no. 1, pp. 48–58, 2003.

[23] T. Espinha, "Web API Responses - MOBILESoft 2015," 01 2015. [Online]. Available: http://dx.doi.org/10.6084/m9.figshare.1284424

[24] E. Babbie, *The practice of social research, 11th edn.* Cengage, 2007.

[25] G. Çalıklı and A. Bener, "Influence of confirmation biases of developers on software quality: an empirical study," *Software Quality Journal*, vol. 21, no. 2, pp. 377–416, 2013.

[26] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the android ecosystem," in *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE CS, 2013, pp. 70–79.

[27] S. Blank (YourTrove), "API integration pain survey results," 2011, website last visited September 27, 2013. [Online]. Available: https://www.yourtrove.com/blog/2011/08/11/api-integration-pain-survey-results/

[28] R. Daigneau, *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley, 2011.