

Testing Cross-Platform Mobile App Development Frameworks

Nader Boushehrinejadmoradi, Vinod Ganapathy, Santosh Nagarakatte, Liviu Iftode
Department of Computer Science, Rutgers University
{naderb,vinodg,santosh.nagarakatte,iftode}@cs.rutgers.edu

Abstract—Mobile app developers often wish to make their apps available on a wide variety of platforms, *e.g.*, Android, iOS, and Windows devices. Each of these platforms uses a different programming environment, each with its own language and APIs for app development. Small app development teams lack the resources and the expertise to build and maintain separate code bases of the app customized for each platform. As a result, we are beginning to see a number of cross-platform mobile app development frameworks. These frameworks allow the app developers to specify the business logic of the app once, using the language and APIs of a home platform (*e.g.*, Windows Phone), and automatically produce versions of the app for multiple target platforms (*e.g.*, iOS and Android).

In this paper, we focus on the problem of testing cross-platform app development frameworks. Such frameworks are challenging to develop because they must correctly translate the home platform API to the (possibly disparate) target platform API while providing the same behavior. We develop a differential testing methodology to identify inconsistencies in the way that these frameworks handle the APIs of the home and target platforms. We have built a prototype testing tool, called X-Checker, and have applied it to test Xamarin, a popular framework that allows Windows Phone apps to be cross-compiled into native Android (and iOS) apps. To date, X-Checker has found 47 bugs in Xamarin, corresponding to inconsistencies in the way that Xamarin translates between the semantics of the Windows Phone and the Android APIs. We have reported these bugs to the Xamarin developers, who have already committed patches for twelve of them.

I. INTRODUCTION

Over the last several years, we have witnessed a number of advances in mobile computing technology. Mobile devices are now available in a variety of form factors, such as glasses, watches, smartphones, tablets, personal robots, and even cars. These devices come equipped with powerful processors, ample storage, and a diverse array of sensors. Coupled with advances in operating systems and middleware for mobile devices, programmers can now avail rich programming APIs to build software (“apps”) that leverage these advances in hardware. Modern app markets contain hundreds of thousands of apps, and the number and diversity of apps available to end-users has further contributed to the popularity of mobile devices. These advances in hardware and software have made mobile devices viable replacements for desktop computers.

At the same time, we are also witnessing a fundamental shift in the practice of software development due largely to the dynamics of mobile app development. Until a few years ago, the task of developing software (targeting mainly desktop computers) was mostly confined to teams of software engineers, either in the open-source community or at IT companies. In contrast, it is common today for small teams

or even individuals to build and distribute software via mobile app markets. Such teams, or individuals, may lack the expertise and experience of a large team of developers and often face economic and time constraints during app development. Nevertheless, mobile app development teams aim to maximize revenue by making their apps available on a wide variety of mobile devices, *i.e.*, those running software stacks such as Android, iOS, and Windows. Apps that are available for a wide variety of mobile devices can reach a large user base, and can therefore generate more revenue either through app purchases or via in-app advertisements.

One way to build apps for different mobile platforms is to create customized versions of apps for each platform, *e.g.*, a separate version of the app for Android, iOS and Windows devices. However, this approach leads to multiple versions of the app’s code-base, which are difficult to maintain and evolve over time. Therefore, developers are increasingly adopting *cross-platform mobile app development frameworks*. These frameworks allow developers to program the app’s logic once in a high-level language, and provide tool-support to allow the app to execute on a number of mobile platforms.

There are two broad classes of cross-platform frameworks available today. The first class, which we call *Web-based frameworks*, allows developers to build mobile apps using languages popularly used to build Web applications, such as HTML5, JavaScript, and CSS. Examples of such frameworks include Adobe PhoneGap/Cordova [1], Sencha [7] and IBM MobileFirst [3]. Developers specify the app’s logic and user interface using one or more of the Web-development languages. However, these languages do not contain primitives to allow apps to access resources on the phone, *e.g.*, peripherals such as the camera and microphone, the address book, and phone settings. Thus, Web-based frameworks provide supporting runtime libraries that end-users must download and execute on their mobile devices. Mobile apps interface with these libraries to access resources on the mobile devices—such mobile apps are also popularly called hybrid mobile apps. Web-based frameworks allow developers to rapidly prototype mobile apps. However, these frameworks are ill-suited for high-performance apps, such as games or those that use animation. The expressiveness of the resulting mobile apps is also limited to the interface exported by the runtime libraries offered by the frameworks.

The second class, which we call *native frameworks*, addresses the above challenges. Examples of such frameworks include Xamarin [8], Apportable [2], MD² [6, 31], and the recently-announced cross-platform bridges to be available on Windows 10 [30, 40]. These frameworks generally support a *home platform* and one or more *target platforms*. Developers build mobile apps as they normally would for the home plat-

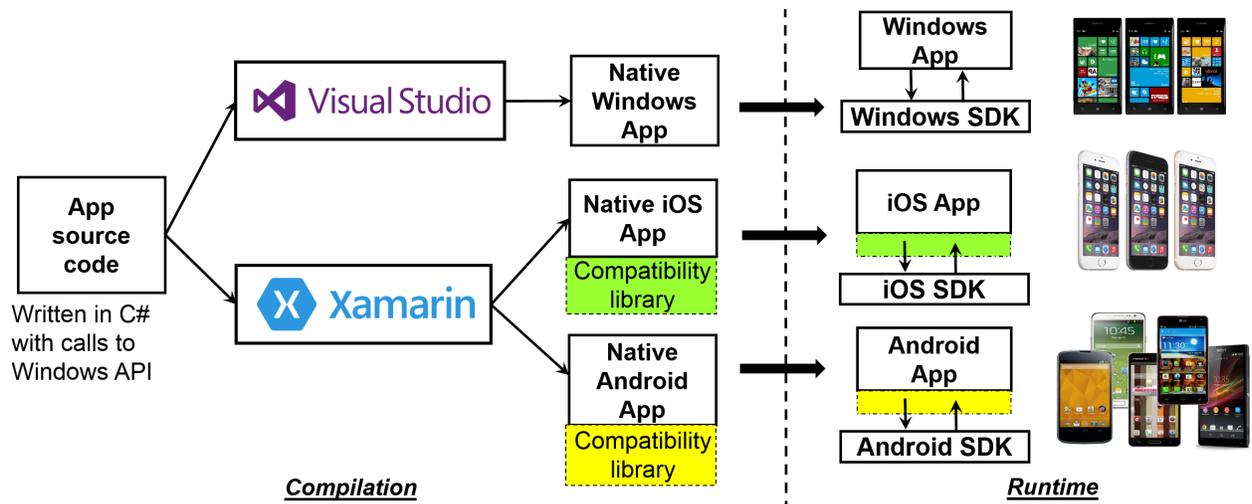


Fig. 1. Overall operation of a cross-platform mobile app development framework, using Xamarin as a concrete example. Developers build apps as they would for the Windows Phone, in C# using calls to the API of the Windows Phone SDK. This code can directly be compiled to Windows Phone apps using the Visual Studio toolchain. Xamarin allows developers to use the same source code to build native Android or iOS apps. Xamarin provides compatibility libraries that translate Windows SDK API calls in the code to the relevant API calls of the underlying Android and iOS SDKs.

form, and leverage the framework’s support to automatically produce apps for the target platforms as well. For example, the home platform for Xamarin is Windows Phone, and developers build apps using C# and the API of the Windows Phone SDK. The Xamarin framework allows developers to automatically build Android and iOS apps using this code base. Likewise, the home platform for Apportable is iOS. Developers build apps using Objective-C and the iOS SDK, and leverage Apportable to produce Android apps from this code base. One of the main highlights of the Microsoft Build Developer Conference held in April/May 2015 was the announcement that the upcoming release of Windows 10 will contain interoperability bridges that allow Android and iOS developers to easily port their apps to the Windows 10 platform [30, 40]. These bridges allow Android (or iOS) apps written in Java (or Objective-C) and programmed to use calls from the Android (or iOS) SDK to transparently execute atop Windows 10 devices. While the technical details of this platform are forthcoming as of this paper’s publication, it is reasonable to assume that the bridges will incorporate a compatibility library to bridge the Android (or iOS) SDK and the Windows 10 SDK. In this paper, we will focus on native frameworks for cross-platform mobile app development. Figure 1 shows the typical workflow of app development using a native framework. We use Xamarin as a concrete example, but the same general workflow applies to all such native frameworks.

When an app developer uses native frameworks, he implicitly expects the apps to behave consistently across the home and target platforms. Realizing this expectation depends to a large extent on the fidelity with which the native framework translates the API calls to SDK of the home platform to the corresponding SDK of the target platform(s). Unfortunately, this translation is a complex task because the platform must correctly encode the semantics of both the home platform and target platform SDK and the relationship between them. This complexity translates into bugs in the frameworks. For example, as of May 2015, Xamarin’s Bugzilla database shows a history of about 7,100 bugs that are related to cross-platform

issues¹, about 2,900 of which are still unresolved (listed as “open” or “new”). Although initial development of Xamarin only started in 2011, its code is based on Mono, which started in 2004 as an open-source implementation of .NET. The fact that such a large number of bugs exist in a mature and heavily-used platform (over 500,000 users) such as Xamarin/Mono point to the complexity of translating between platforms. Other native frameworks are no exception, either. Apportable’s bug database, for instance, shows a history of 820 bug reports, 449 of which are still unresolved.

In this paper, we develop an approach to test native frameworks. Specifically, we aim to discover cases where the behavior of the application on the home platform is inconsistent with the behavior of its counterpart on a target platform. Our approach is based on *differential testing* [33]. We generate random test cases (using methods described in prior work [36]), which in our case are mobile apps in the source language of the home platform. We then use this code to produce two versions of the app, one for the home platform, and one for the target platform using the native framework. We then execute the apps and examine the resulting state for inconsistent behavior. When two versions of the app are produced from the same source code, any differences in the behavior across the versions are indicative of a problem either in the home platform’s SDK, the target platform’s SDK, or the way the native framework translates between the two SDKs.

To realize this approach, we must address two issues:

(1) Test Case Generation. The key research challenge in generating effective test cases is that the space of valid programs that we can generate as test cases is essentially unbounded. While we could sample from this space, the probability that these test cases will induce inconsistent behavior is low.

¹There are a total of about 20,300 bugs in the database, related to various related products offered by Xamarin, e.g., the profiler, the IDE environment, etc. We do not count those bugs because they are not directly related to cross-platform issues.

To address this challenge, we observe that the main difficulty in building cross-platform mobile app development tools is *translating between the semantics of the SDKs of the home and target platforms*. Our test-case generator therefore produces programs that contain random sequences of invocations to the home platform’s SDK. We then observe whether the resulting apps on the home and target platforms behave consistently. By focusing on the SDK alone, our approach narrows testing to the most error-prone components of the cross-platform frameworks.

(2) Test Oracle Design. Each of our test cases is compiled into a full-fledged app, one each for the home and target platforms. When we run the corresponding apps, the test oracle must observe their behaviors to identify inconsistencies. The main research challenge here is in defining a suitable notion of “behavior” that can be incorporated into our test oracle.

We address this challenge by observing all data structures that are reachable from the variables defined in the test cases. We serialize these data structures into a standard format, and compare the serialized versions on the home and target platforms. Assuming that the state of the home and target platforms is the same before the test cases are executed, the final state in each platform after the test cases have been executed must also be the same. If not, we consider this inconsistent behavior and report an error.

We have prototyped this approach in a tool called X-Checker, which we have applied to test the Xamarin framework using Android as the target platform. Using X-Checker, we have found 47 inconsistencies, which corresponded to bugs either in Xamarin or the Microsoft SDK (we have reported these to Xamarin or Microsoft). To date, 12 of these bugs have also been fixed in the development branch of Xamarin [9–19] and others are still open.

To summarize, our contributions are:

- We initiate the study of cross-platform mobile app development frameworks, and present an analysis of the kinds of bugs that may arise when these frameworks translate between the semantics of the programming interfaces of two different platforms (Section III).
- We present the design of X-Checker, a testing tool for cross-platform frameworks that uses random differential testing to expose bugs in these frameworks (Section IV). We also present a number of practical challenges that we had to overcome in the implementation of X-Checker (Section V).
- We show the effectiveness of X-Checker by applying it to Xamarin. Specifically, X-Checker tests Xamarin’s fidelity as it translates between the Windows Phone and Android platforms. To date, X-Checker has found 47 bugs, 12 of which have been fixed after we reported them (Section VI).

II. BACKGROUND ON NATIVE FRAMEWORKS

In this section, we provide background on native frameworks using Xamarin as a concrete example. Xamarin allows the development of native mobile apps for multiple platforms while aiming to maximize code-reuse across platforms. Developers using Xamarin target their apps to its home platform, Windows Phone, and can re-use much of the same code to build native apps for iOS, Android, and Mac. In this section,

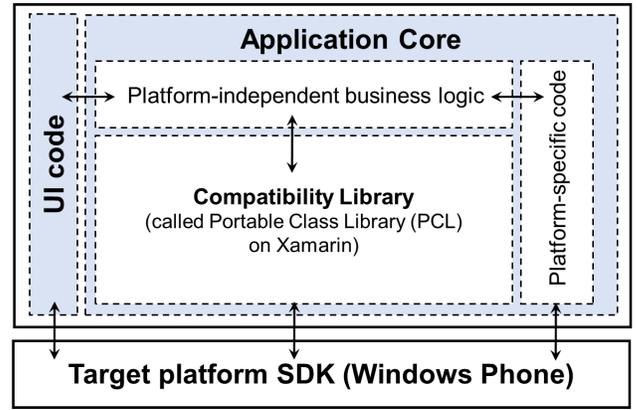


Fig. 2. Structure of a cross-platform app written using Xamarin.

we discuss the structure of a cross-platform app written using Xamarin, and discuss the techniques that Xamarin uses to allow app logic and data storage code to be written once and reused across platforms.

A developer using Xamarin can build apps in C#, using features such as generics, Linq and the parallel task library. The developer splits the app into two logical pieces (Figure 2): *the application core*, which encodes the business logic, and contains code that is common across all platforms, and *user interface (UI)*, which is written for each platform and uses the native UI features of that platform, *e.g.*, buttons, widgets, and the overall look and feel of the specific platform. The developer implements the UI layer in C# as well, using native UI design tools such as `Android.Views`, `MonoTouch.UIKit` for iOS, and XAML, Silverlight and Metro APIs for Windows Phone. The functionality and layout of the UI elements can be controlled by the business logic in the application core, *e.g.*, in determining which button triggers what functionality in the app. Xamarin is built atop the Mono .NET framework [5], which provides the core cross-platform implementation of Microsoft’s .NET framework. C# source code can be compiled with Xamarin’s compiler to produce a native iOS app, or an Android app with integrated .NET runtime support. In this case, the C# code is compiled to an intermediate language, and packaged with MonoVM configured for just-in-time compilation on Android.

Xamarin aims to provide support to developers to minimize the amount of platform-specific code that is needed to port an app across platforms. To achieve this goal, one of the main components of the core of a Xamarin-based app are cross-platform compatibility libraries, also called *portable class libraries*, or *PCLs* in Xamarin, a technology originally developed by Microsoft. On Visual Studio and other Microsoft environments, a PCL is a special type of a project that allows developers to write code and produce libraries that can be shared across multiple platforms, such as iOS, Android, and Windows Phone. To support this, PCLs export an interface of methods and properties that are portable across platforms, and developers program to this interface. The app developer encodes platform-independent business logic by programming to this interface. The PCL provides forwarding stubs that ensures that calls to methods or property accesses are routed to the correct underlying platform libraries at runtime. The developer

of the PCL typically identifies the interface by choosing a set of target platforms that the PCL will support. Because different platforms provide implementations of differing subsets of the base .NET class library, the PCL interface is typically restricted to the common .NET functionality that is supported by all the target platforms.

PCLs play a key role in Xamarin because they serve as the compatibility layer between two different platforms. As previously mentioned, about Xamarin’s BugZilla database lists about 7,100 that are related to PCL. Despite the functionality provided by the PCLs, some platform-dependent business logic may be necessary in the application core. For example, PCLs are still in active development, and if the app developer wishes to use features that are not currently supported by the PCL, he has to do so by writing platform-specific code, called *shared assets* on Xamarin. It is possible to write this code once and compile it for all desired target platforms using compiler or pre-processor directives (e.g., code specific to Android or iOS would be guarded using a directive such as `#ifdef ANDROID` or `#ifdef iOS`, respectively). Naturally, the goal of projects such as Xamarin is to increase the coverage provided by their PCLs, so as to minimize the amount of code that must be written as shared asset projects.

In addition to the application core, the app also includes UI code. Currently, UI code is largely platform-specific because UI elements, e.g., the look and feel of buttons and widgets, are customized to specific mobile platforms. Nevertheless, there are ongoing efforts such as `Xamarin.Forms` to even minimize the amount of platform-specific UI code.

In this paper, we are primarily concerned with testing the functionality of the PCLs on Xamarin that provide support for platform-independent app code. Therefore, the test cases generated by X-Checker only target the PCL interface. Our test cases do not directly target the platform-specific UI code. However, note that many aspects of the layout and functionality of the UI are controlled by the business logic, which interacts with the target platform’s SDK via the PCLs. Therefore, by testing the functionality of the PCLs, we indirectly test the overall functionality of the app’s execution on the target platform (including its UI).

III. INCONSISTENT BEHAVIOR

In this section, we present a few motivating examples of real inconsistencies that X-Checker found in Xamarin. For our examples, we use Android as the target platform; the default home platform is Windows phone. We use these examples to motivate some design features of X-Checker, and classify the types of inconsistencies that it can identify.

Figure 3 shows a test case generated by X-Checker. This code is in C# and uses classes and methods from the Windows Phone SDK. In this test case, the code first creates two objects, `base` and `exp`, from the `System.Numerics.Complex` class, and initializes them to `0+0i` and `1+0i`. On line (8), it uses the `Complex.Pow` operation to raise `base` to the power of `exp`.

We used the Visual Studio toolkit and the Xamarin framework to produce a Windows Phone app and an Android app, respectively, and ran the apps on the corresponding platforms. Both apps execute and return `SUCCESS`. However, in

```
(1)using System.Numerics;
(2)Serializer serializer; // Serializer is a data structure serializer.
(3)public class TestCase {
(4)    public static int TestMain (MyFileIO serialStream, MyFileIO logStream) {
(5)        try {
(6)            Complex base = new Complex(0,0);
(7)            Complex exp = new Complex(1,0);
(8)            Complex res = Complex.Pow(base,exp);
(9)            serialStream.append(base.GetType().FullName, serializer.serialize(base));
(10)           serialStream.append(exp.GetType().FullName, serializer.serialize(exp));
(11)           serialStream.append(res.GetType().FullName, serializer.serialize(res));
(12)           return SUCCESS;
(13)        } catch (System.Exception e) {
(14)            logStream.append(e.GetType().FullName);
(15)            return EXCEPTION;
(16)        }
(17)    }
(18)}
```

Fig. 3. A test case that illustrates inconsistent handling of the semantics of the Windows Phone SDK. The values of `res` are different in the Windows Phone and the Xamarin-produced Android versions of this code.

Type	Platform 1	Platform 2	Consistency checks	Example
(1)	✓	✓	Check app state	Figure 3
(2)	×	×	Check exception code	Figure 5
(3)	✓	×	Always inconsistent	Figure 6

Fig. 4. Different ways in which a test case produced by X-Checker can exhibit inconsistent behavior when executed on two platforms. ✓ denotes that the test case returns `SUCCESS`, while × denotes that the test case returns `EXCEPTION`.

the Windows Phone app, the value of `res` is `0+0i`, while on the Android app, the value is `NAN` (not a number). This is clearly an inconsistency in the way the two apps handled the semantics of the `Complex.Pow` operation. Since we reported this bug on Xamarin’s BugZilla forum, it has been fixed in the master branch for the next release [17].

In this example, eliciting the inconsistent behavior between the Windows Phone and the Android versions of the app requires the calls on lines (6)-(8), with the corresponding data dependencies. To systematically uncover more examples of such inconsistencies, X-Checker must therefore generate many more such test cases by systematically invoking methods from the API of the Windows Phone SDK with suitable arguments.

When the apps produced from these test cases are executed on their corresponding platforms, inconsistent behavior may manifest itself in one of three ways (Figure 4). The first way, as illustrated in the example in Figure 3, is where the test case returns `success` on both platforms, but the resulting state is different. Such inconsistencies are latent in the state of the apps, in this case, the values of the objects, and are not visible unless this state is made explicit and compared across the two versions.

X-Checker achieves this goal by *serializing* all objects that are reachable from the variables that are in scope within the source code of the app. Lines (9)-(11) in Figure 3 show the objects being serialized and appended to a log. X-Checker compares the logs produced by the Windows Phone and the Android versions of the apps to identify inconsistencies. In this example, serializing the `Complex` object simply prints its value to the log. However, X-Checker’s serializer supports arbitrary data types, and serializes them in a custom format. The serializer itself is written in C#, with calls to the Windows Phone SDK, and is included as a library within the native app. As with all our test cases, we use Xamarin to produce the Android version of the serialization library. Because X-Checker’s test cases include calls to the serializer in the source code of the test case, we expect the serialized versions of similar objects to also be similar on the Windows Phone and Android versions.

```

(1) public class TestCase {
(2)     public static int TestMain (MyFileIO serialStream, MyFileIO logStream) {
(3)         try {
(4)             string s = "test";
(5)             int32 index = -1;
(6)             Double val = System.Globalization.CharUnicodeInfo.GetNumericValue(s, index);
(7)             return success;
(8)         } catch (System.Exception e) {
(9)             logStream.append(e.GetType().FullName);
(10)            return EXCEPTION;
(11)         }
(12)     }
(13) }

```

Fig. 5. A test case that triggers an inconsistent exception behavior. In the Windows Phone version of this code, line (6) throws a `System.ArgumentOutOfRangeException`, while on the Xamarin-produced Android version, it throws a `System.IndexOutOfRangeException`. For brevity, we have omitted some code, such as calls to the serializer.

A second way for inconsistencies to manifest is when a test case returns `EXCEPTION` on both platforms, but the exceptions thrown are different on both platforms. Figure 5 illustrates a test case in which this scenario occurs. The call on line (6) throws an exception because the value of `index` is negative. However, the Windows Phone version throws a `System.ArgumentOutOfRangeException`, while the Android version throws a `System.IndexOutOfRangeException`. In this case, the Windows Phone and Android versions are inconsistent in the way they handle the semantics of the `GetNumericValue` method. X-Checker therefore logs the exception code, and compare it across executions of the apps on the two platforms. This bug has also been fixed in the master branch for the next release [9].

In cases such as these, where the test cases on both platforms throw exceptions, the logs only contain the exception code. In particular, the logs do not contain the serialized data structures because the calls to the serializer appear before the `return success` statement, and the exception was raised before control reached the calls to the serializer. It may be possible that both the Windows Phone and the Android versions throw the same exception code, but the state of the data structures in the apps may have diverged before the code that raised the exception was executed, which is also an example of inconsistent behavior. As will be clear when we discuss X-Checker’s approach to test case generation in Section IV, X-Checker would have also identified the divergence of state. In particular, X-Checker uses an iterative test case generation algorithm that preserves the following property: any prefix of a method sequence in a test case generated by X-Checker is also a test case that would have been generated by X-Checker in a previous iteration. Therefore, if the state is inconsistent after a call sequence preceding the exception-generating method, it would have been identified as an inconsistency when the shorter method sequence was used as a test case.

Note that in Figure 5, the test case executes the code and catches a generic `System.Exception`. In practice, it may be that a developer writing a Windows Phone app, familiar with the Windows Phone SDK, may write this code to catch a `System.ArgumentOutOfRangeException`. If he uses Xamarin to produce an Android app, it is possible for the the inconsistent behavior to manifest itself in one of the other two forms shown in Figure 4.

The final possibility for an inconsistency is when a test case returns `SUCCESS` on one platform, and `EXCEPTION` on the other. Figure 6 shows an example of such a test case. The

```

(1) using System.Xml;
(2) public class TestCase {
(3)     public static int TestMain (MyFileIO serialStream, MyFileIO logStream) {
(4)         try {
(5)             NameTable nt1 = new NameTable();
(6)             NameTable nt2 = new NameTable();
(7)             XmlNamespaceManager nsMgr = new XmlNamespaceManager(nt2); ...
(8)             XmlParserContext xpctx = new XmlParserContext(nt1, nsMgr, ...); ...
(9)             return success;
(10)        } catch (System.Exception e) {
(11)            logStream.append(e.GetType().FullName);
(12)            return EXCEPTION;
(13)        }
(14)    }
(15) }

```

Fig. 6. A test case that triggers an exception in the Windows Phone version. The constructor on line (8) throws an `XmlException` because `nsMgr` is independent of `nt1`. This test case executes without throwing an exception on the Xamarin-produced Android version.

`XmlParserContext` constructor in line (8) expects its second argument (`nsMgr`) to be created from the first argument (`nt1`). However, in this case, `nsMgr` is created from another object `nt2`. As a result, this constructor call must throw an `XmlException` according to Microsoft’s documentation, and it does on the Windows Phone version. However, on the Android version the constructor executes without throwing an exception. As with the previous two bugs, this one also has been fixed by Xamarin developers after we reported it [11].

IV. DESIGN OF X-CHECKER

X-Checker aims to find bugs in Xamarin by generating apps, executing these apps on Windows Phone and Android, and looking for inconsistencies in them. Thus, X-Checker’s design consists of two parts, the test case generator and the inconsistency checker.

Test Case Generation. X-Checker generates test cases that exercise the programming API used by Windows Phone developers. As illustrated in Section III, each test case is a sequence of method calls to this API. The arguments to these calls are either values with primitive data types, or references to objects constructed and modified by method calls appearing earlier in the sequence. The main challenge is to generate meaningful method sequences that are also effective, *i.e.*, the test case generator should be able to elicit error cases in Xamarin without generating too many test cases.

This problem has been investigated in the past in the context of generating unit tests for object-oriented programs, and tools such as JCrasher [27] and Randoop [36] implement such test case generation. In particular, Randoop uses a *feedback-directed approach* to random test generation and is the basis for X-Checker’s test generator as well. We now briefly describe Randoop’s (and therefore X-Checker’s) approach to test generation.

The test generator accepts as input a list of classes to be tested, a set of filters and contracts (which are sanity checks to be performed), and a timeout. Intuitively, the test generation algorithm iteratively “grows” method sequences from previously-generated shorter sequences. Suppose that the test generator has already generated a set of method sequences as valid test cases. During each iteration, the test generator picks a method $m(T_1, \dots, T_n)$ at random from the list of classes provided to it as input, and “extends” the existing method sequences with a call to m (*e.g.*, one way to “extend” is to append m to the end of the sequence). If the parameters of m are all of primitive type, then the test generator randomly selects the values of these parameters from a pool of acceptable

values. If the parameter is a reference to an object, then the test generator uses an object of suitable type created by a method in the sequence that `m` just extended (or passes a `NULL` reference). X-Checker then wraps this method sequence with template code to serialize data structures and to catch exceptions, as shown in the examples from Section III, to produce the test cases.

The test generator then executes the newly-generated test sequences looking for violations of filters and contracts. These are sanity checks that look for common error cases, such as test cases that throw an exception, or those that violate certain invariants (e.g., `o.equals(o)` not returning `TRUE`). Test sequences that violate these sanity checks are discarded, and the remaining test cases are added to the set of valid test cases, to be extended in future iterations. This process continues till the specified timeout has expired. This iterative approach is key to generating effective test cases. It ensures that every prefix of a valid test sequence is also valid, and that test sequences that violate simple sanity conditions (e.g., those that throw an exception) are never extended.

Serializing State and Checking Inconsistencies. For the test cases generated using the approach above, X-Checker produces a pair of apps for Windows Phone and Android. It executes them atop these platforms to observe inconsistent behavior. We now discuss the design of the serializer, which helps identify inconsistencies when both apps return `SUCCESS`, i.e., the first case in Figure 4. The other two cases are straightforward and we do not discuss them further.

The serializer recursively traverses object references to create serialized representations. Intuitively, a serialized representation is a set of (key, value) pairs. The key is the name of a public field of the object. For fields of primitive type (e.g., `bool`, `int`, `String`), the value is simply the actual value of the field. For fields that are themselves object references, the value is a serialized representation of that object. The example below shows the serialized representation of a linked list with two entries. The data field of the entries store 1 and 2, respectively.

```
{{("data", 1), ("next", {("data", 2), ("next", NULL)})}}
```

X-Checker's serializer uses the `Json.NET` [4] library, which optionally supports the ability to serialize cyclic data structures. It does so by keeping track of object references using an additional identifier. However, in our experience, the random test cases that we generate do not produce cyclic heap data structures. We therefore did not enable support for serializing cyclic data structures in our prototype, and the serialized object representations are tree-structured as a result. Note that in the unlikely case that a test case does produce a cyclic data structure, our serializer would infinitely loop—a situation we have not encountered to date in our experiments.

X-Checker identifies inconsistencies by comparing serialized representations of objects on the home and target platforms. Comparison proceeds recursively in a bottom-up fashion. All the (key, value) pairs storing primitive types must match, and the tree-structure of the serialized representation must be the same, i.e., the same keys on both platforms at each level of the tree. Any mismatches indicate inconsistent state. In most of the bugs that we found, the mismatches were because

the values diverged (e.g., the complex number example in Figure 6). However, we also found cases where the fields in the objects were different on Windows Phone and Android because a field that was declared to be `public` on Windows Phone was a `private` field in Android, and therefore not listed in the serialized representation.

As previously discussed, the feedback-directed approach to test case generation does not extend any method sequences that violate its filters and contracts, e.g., sequences that throw an exception when executed. While Randoop was originally designed for unit-testing object-oriented programs, X-Checker extends it for cross-platform differential testing. For practical reasons described in Section V, X-Checker first executes the test case generator on one platform, where it uses the iterative approach to output test cases. It then executes these test cases on the target platform (Android). Thus, X-Checker's test cases also preserve the property that only non-exception-generating test cases are extended in the iterative process.

However, because X-Checker generates all the test cases on the home platform before executing them on the target platform, even those test cases that return success but produce inconsistent serialized state across the two platforms are extended during test generation. As a result, it is possible that multiple test cases produced by X-Checker may report the same inconsistency.

Discussion. Differential testing offers an attractive property. If a test case is executed on two API implementations with the same initial state, any inconsistency in the final states indicates a problem in at least one of the API implementations. That is, differential testing *does not produce false positives*.

However, in practice, it is possible that an inconsistency does not always correspond to a problem. In our experiments, we found that such a situation could arise because of any one of a small number of reasons. First, some API methods, such as those from `System.Random` and `System.Time`, invoke platform-specific features and return different values when invoked on different platforms. For example, the `System.Net.Cookie()` constructor initializes `Cookie.TimeStamp` with the current system time. Unless the emulation environments that run the apps for both the home and target platforms are synchronized, this call will return different values. Second, for some methods, such as `Object.GetHashCode`, the documentation specifies that the behavior of the method varies across platforms. That is, the `HashCode` of an object can be different on the home and target platforms even if the serialized representations of the object are the same on both platforms. A third source of false positives was because the Mono runtime included in a Xamarin-produced Android app uses Mono Assemblies as its libraries. These libraries have different metadata information than their Windows Phone counterparts, and any calls that access this metadata will result in inconsistent serialized state.

Fortunately, it is relatively easy to filter out test cases that can potentially lead to such false positives. We simply integrate filters that prevent the test generator from producing method sequences that contain method calls or field references that can potentially trigger such false positives. Thus, with just a few filters to eliminate the causes above (see Figure 7), we were able to eliminate false positives, thereby ensuring that all

Filtered classes: All methods/variables of this class filtered.	
System.Random	— Members return random values
Filtered methods: Methods cannot appear in test cases.	
System.Type GetType()	— Return value may include information from the underlying C# assembly, which is not uniform across platforms
System.Int32 GetHashCode()	— Documentation specifies that hash code of similar objects need not be similar across platforms
Filtered constructors: Constructor cannot appear in test cases.	
System.Xml.UniqueId()	— Returns a unique GUID, which is not guaranteed to be consistent across platforms
Filtered fields/properties: Cannot be accessed in test cases.	
System.Net.Cookie.Timestamp	— Value relies on system time at object creation
System.DateTimeOffset.Now	— Value relies on system time
System.DateTimeOffset.UtcNow	— Value relies on system time
System.DateTime.Now	— Value relies on system time
System.DateTime.UtcNow	— Value relies on system time
System.DateTime.Today	— Value relies on system time
System.Exception.HResult	— Value identifies an exception, but documentation is not conclusive about whether value is consistent across platforms

Fig. 7. Filters used by X-Checker to avoid generating test cases that produce false positives.

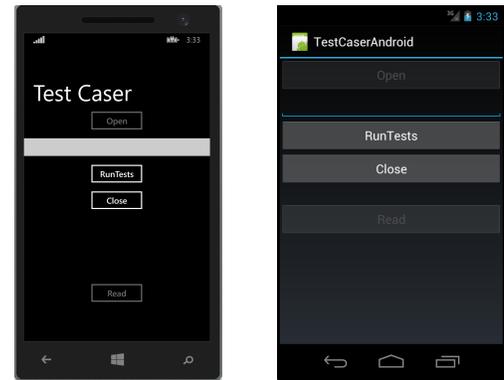
inconsistencies reported by X-Checker indeed correspond to real bugs. Note, however, that as with most other testing tools, our differential testing approach does have false negatives—*i.e.*, it is not guaranteed to find all possible inconsistencies.

V. PRACTICAL CONSIDERATIONS

In this section, we discuss a few practical issues that we had to address as we built X-Checker.

Where to Generate Test Cases? The first practical consideration that we addressed was the question of which platform to use to execute our test generator. One possibility was to use Windows Phone, Xamarin’s home platform. This requires us to execute X-Checker directly on a device or emulator running Windows Phone. However, we found that the environment on such devices and emulators was somewhat awkward to use during active development of X-Checker, *e.g.*, to debug any issues that arose. We therefore decided to develop and execute X-Checker on a desktop version of Windows (8.1). Our hypothesis was that the desktop and phone versions would be largely similar because they use the same .NET code base, and as a result, we could use the desktop to generate the test cases and execute them as apps on Windows Phone and Android devices.

This approach eased development of X-Checker, and for the most part our hypothesis about the equivalence of the desktop and phone version of Windows was correct. However, we found (and reported to Microsoft) a case where the desktop and phone versions diverged in their semantics. In particular, the public property `CurrencyDecimalDigits` of the class `System.Globalization.NumberFormatInfo` is required to be a read-only field according to MSDN documentation. While the read-only property holds in the desktop version, the property is mutable in the Windows Phone version. We also found, using differential testing against Android, a case where both the desktop and phone version of Windows incorrectly implement the documented semantics for a given property, while Android’s implementation followed Microsoft’s documentation. In particular, the property `WebName` of `System.Text.Encoding.BigEndianUnicode` must have the



(a) Windows Phone version (b) Android version

Fig. 8. Screenshots showing the UIs of the test case apps on Windows Phone and Android.

value UTF-FFFE according to Microsoft’s documentation, but the desktop and phone version of Windows return UTF-16BE.

PCLs and Test Generation. A second issue that we had to address was the integration of X-Checker’s test generator and PCLs. As previously discussed, X-Checker’s test generator produces test cases for a given input set of classes. It uses reflection to identify public methods from those classes, the data types of their arguments and the other properties of these classes, and uses this information in its test generation algorithm.

However, PCLs pose a unique problem when used with X-Checker’s test generator. Recall that PCLs define the interface against which developers can build their applications without concerning themselves with cross-platform portability issues. PCLs enable this feature by transparently acting as forwarders—they route calls from the application layer to the corresponding library in the platform on which the application is loaded. Thus, PCLs usually do not contain any of the executable code of the classes for which they act as an interface, and merely contain forwarding stub code. As a result of this feature, when X-Checker’s test generator is provided with a set of PCL classes as input, it is unable to use reflection to fetch the complete set of public methods, data types and properties of the classes for which the PCL acts as a forwarder.

To address this issue, we had to extract the information required by X-Checker’s test generator by loading PCLs in a non-executable *reflection-only* mode. In this mode, PCLs are not executable, but can be queried using reflection and return metadata by accessing the corresponding classes on the platform. We then re-load the PCLs in executable mode, and use the metadata to iteratively generate and execute test cases.

How to Package Test Cases? Finally, we also had to address the issue of how to package up the test cases for execution on the Windows Phone and Android platforms. Each test case is packaged as a separate class that can be instantiated and executed. As discussed above, we run the iterative test generation algorithm on the desktop version of Windows. As a result, we have all the test cases available for batch execution on the mobile platforms.

We package all the test cases into a single app each for execution on the two mobile platforms. Both Windows Phone

and Android require apps to define a UI. We wrote this UI and the code to interface with the file system (to store the logs generated when test cases are executed) within a platform-specific presentation layer, and packaged up the test cases as platform independent code to be cross-compiled by Visual Studio and Xamarin. All the test cases generated by X-Checker can be invoked at the press of a single button on the app. Figure 8 shows the UIs of the Windows Phone and Android versions of these apps. The UIs of these apps look rather different—each app uses buttons and icons unique to the corresponding mobile platform. However, because our test cases focus only on the platform-independent PCL classes, the differences in the UI state do not manifest as divergent state (and therefore as inconsistencies) during the execution of the test-cases.

VI. EXPERIMENTAL RESULTS

Setup. For our experimental evaluation, we used Xamarin.Android version 4.16.0, business edition. We chose Windows 8.1 as the home platform, and Android 4.0.3 (API level 15) as the target platform. As discussed in Section V, we generate test cases on a desktop version of Windows, and then run these cases on Windows Phone and Android platforms. Both the phone and desktop version of Windows use .NET version 4.5.51641. We use Visual Studio Ultimate 2013 version 12.0.30501 as the IDE to compile our test cases. This environment supports a package that integrates the tools for Windows Phone 8.1 into the controls of Visual Studio. We also use the same development environment to build the Android version using Xamarin. In particular, we use the Xamarin 3.5.58.0 extension to enable development for Xamarin.Android within Visual Studio.

We use emulators to mimic Windows Phone and Android devices. Microsoft offers a few pre-configured emulation environments for Windows Phone: our experiments use Emulator 8.1/WVGA-4inch/512MB configuration. We configured the Android emulator to match the hardware configurations of the Windows Phone emulator.

Examples of Inconsistent Behavior. Figure 9 presents the results of our experiments. To date, we have used X-Checker to generate 22,465 test cases, which invoke 4,758 methods implemented in 354 classes across 24 Xamarin DLLs. In all, we observed 47 unique instances of inconsistent behavior across Windows Phone and Android. The results also show a detailed breakdown of these inconsistencies by category, where the type of the inconsistency is as defined in Figure 4.

In most cases, we were quickly able to quickly confirm using MSDN and Xamarin documentation that the inconsistency was indeed a bug in Xamarin. For each type of inconsistency in Figure 4, the test cases that induced them and the inconsistent results they produced were largely similar to the examples described in Section III. We now discuss a few interesting examples of inconsistencies that we found.

(1) *Precision in math libraries.* We observed two inconsistencies that were related to precision with which math libraries used rounding and precision to represent numbers. In one test case, a call to `System.Math.IEEEERemainder(double x, double y)` was invoked with $x=1.49089193085384E-81$ and $y=2.22275874948508E-162$. The Windows Phone ver-

sion returns a result of $3.33639470813326E-163$, while the Android version produced by Xamarin returns 0 .

The second test case was a method sequence with two calls. The first call, `System.Math.Round(Decimal d, int i, MidpointRounding mode)` was invoked in the test case as `System.Math.Round(2, 3, ToEven)`. According to the documentation, this call returns the value d rounded with i fractional digits in the given mode. The Windows Phone version returns 2.000 while the Android version returns 2 . While these are equivalent if used in a mathematical calculation, the second call in the test case converted them to strings using `System.Convert.ToString`, which resulted in inconsistent serialized state. These examples highlight inconsistent handling of floating point arithmetic across platforms.

(2) *Ambiguous documentation of exception semantics.* We observed one test case where different exceptions were raised for the same failing method call because of ambiguity in the semantics of the exception to be raised. According to documentation, the `NameTable.Add(Char[] key, int start, int len)` call can throw two types of exceptions. It throws `IndexOutOfRangeException` when any one of these three conditions is met: $0 > \text{start}$, $\text{start} \geq \text{key.Length}$, or $\text{len} \geq \text{key.Length}$. It throws `ArgumentOutOfRangeException` if $\text{len} < 0$.

In one of our test cases, the values of `start` and `len` were such that $0 > \text{start}$ and $\text{len} < 0$. For this test case, both the Windows Phone and desktop versions threw `IndexOutOfRangeException` whereas Xamarin's Android code threw `ArgumentOutOfRangeException`. Although both implementations are correct, the documentation must be clarified to remove this ambiguity.

(3) *Documented deviations of behavior.* For some methods, documentation specifies that the behavior of the method will vary across platforms. Thus, the Xamarin and .NET implementations of these methods need not be similar. For example, consider the constructor for the `UriBuilder` class. The documentation specifies that if this class is implemented in a PCL, then if an invalid URI is provided as the string argument to the constructor, it must throw a `FormatException` instead of a `UriFormatException`.

We also observed examples where the deviations in behavior were not specified formally in the documentation, but were known to the developers of the platform. One such example is the method `ReadContentAsString` from the class `XmlDictionaryReader`. When included in a test case, this method showed inconsistent behavior across the Windows Phone and Android versions. However, when we tried to identify the cause of this bug by examining the source code of the Mono platform (which Xamarin extends to provide a cross-platform implementation of .NET), we found that it was marked with a `MonoTODO` attribute, indicating a known issue with its implementation.

We were not aware of these documented deviations in behavior when we tested the methods using X-Checker, and the resulting differences were reported as inconsistencies. However, because the documentation (or code comments) did specify that the inconsistencies were expected across platforms, we do not count these as bugs (and therefore they do not factor into the 47 inconsistencies reported in Figure 9). Nevertheless, we feel that for such methods, the deviations

Library	#Classes	#Methods	#Tests	#Inconsistencies (by type)		
				Type 1	Type 2	Type 3
Microsoft.CSharp	6	56	1,848	0	0	0
Microsoft.VisualBasic	17	127	613	0	0	0
System.Collections.Concurrent	10	77	349	0	0	0
System.Collections	29	172	532	0	0	0
System.ComponentModel	5	4	1,578	0	0	0
System.Dynamic.Runtime	29	201	790	1	0	0
System.Globalizatiion	14	288	567	3	3	0
System.Linq	5	172	591	0	0	0
System.Linq.Expressions	44	633	590	1	0	1
System.Net.Http	44	524	746	3	0	3
System.Net.NetworkInformation	1	1	1	0	0	0
System.Net.Primitives	13	105	956	0	1	1
System.Net.Requests	10	122	1,269	0	0	0
System.ObjectModel	16	52	1,573	0	0	0
System.Resources.ResourceManager	4	28	1,333	0	1	0
System.Runtime.Extensions	12	409	946	3	1	1
System.Runtime.Numerics	2	170	1,514	3	0	2
System.Runtime.Serialization.Json	4	37	1,642	1	0	0
System.Runtime.Serialization.Primitives	13	86	1,387	1	0	1
System.Runtime.Serialization.Xml	14	342	420	1	3	1
System.Text.Encoding	5	66	940	1	0	0
System.Text.RegularExpressions	10	103	848	0	0	0
System.Xml.ReaderWriter	24	346	820	2	3	3
System.Xml.XDocument	23	637	612	0	1	1
Total	354	4,758	22,465	20	13	14
				47		

Fig. 9. Summary of inconsistencies found by X-Checker in various Xamarin libraries. This table shows the number of classes in each library and the number of methods in these classes. It also shows the number of test cases that X-Checker generated for those libraries, and the number of cases of inconsistent behavior across platforms. These inconsistencies are sorted by type, as defined in Figure 4.

of behavior should be encoded more explicitly (*e.g.*, as pre-conditions) rather than being latent in documentation or in code comments.

Performance. Finally, we report the time taken to run test cases on our experimental setup. We ran the Windows Phone and Android emulators on a desktop system running Windows 8.1 professional edition, and equipped with an Intel Core-i7-3770 running at 3.4GHz, 16GB of RAM. We created an app that packaged 1000 randomly-generated tests and ran the Windows Phone and Android versions of this app on both emulators. The Android emulator took 29.1 seconds to run the app, while the Windows Phone emulator took 2.7 seconds. The Android emulator is much slower because it emulates the ARM architecture atop our Intel platform. In contrast, the Windows Phone “emulator” uses hyper-V and is implemented as a virtual machine. The tests used to report the results in Figure 4 were generated by analyzing each library separately. We configured our test generator to emit test cases until a timeout of 300 seconds was reached for each library being analyzed.

VII. THREATS TO VALIDITY

Our results show the effectiveness of using random differential testing at finding bugs in native app development frameworks. We now summarize the threats to the internal and external validity of our results.

The main threat to internal validity is in determining whether an inconsistency discovered by X-Checker is indeed a symptom of a bug in Xamarin. Although an inconsistency manifests itself in one of the three different ways outlined in Figure 4, it may be the result of using a method with a documented difference in behavior across platforms. We addressed this threat in two ways. First, as discussed in

Section IV, we created filters for methods with documented deviations of behavior, so false-positive-generating test cases are not produced. Second, we studied the results of X-Checker to understand the cause of the inconsistency. In some cases, this study lead us to a sentence in the documentation or code comments where the inconsistency was documented (as discussed in Section VI). We did not include these inconsistencies in our overall count shown in Figure 9, and reported the other inconsistencies to the Xamarin BugZilla forum.

A second threat to internal validity is the “seriousness” of the bugs found by X-Checker—*i.e.*, does an inconsistency lead to a serious error in the functioning of an app, or is it just a minor annoyance? Unfortunately, this aspect is much harder to evaluate. Our take on the issue is that an inconsistency is indeed a bug that must be fixed (or suitably documented). However, the fact that 12 (over 25%) of the inconsistencies that we found lead to bug-fixes within days of our reports shows that Xamarin developers did perceive the inconsistencies as being significant.

The main threat to external validity is the ability of our approach to generalize to other native frameworks, or even other aspects of Xamarin (*e.g.*, the compatibility libraries used to translate between Windows Phone and iOS). We currently do not have experimental data to answer such questions. Nevertheless, our results with Xamarin PCLs for Android indicate that inconsistencies arise because of the challenges involved in translating the semantics of two different mobile platforms. In particular, an analysis of our results does not indicate that the kinds of inconsistencies we found are symptomatic of problems either in Windows Phone or Android alone. Therefore, we hypothesize that random differential testing of

other native frameworks is quite likely to find similar bugs in them as well.

VIII. RELATED WORK

Testing Cross-platform Apps. To our knowledge, our work is the first on testing cross-platform mobile app development frameworks. However, there has been prior work on testing cross-platform apps themselves. The most relevant projects in this area are X-PERT [25] and FMAP [26]. Both projects start with the observation that an increasing number of Web applications create customized views of Web pages, each optimized for different platforms, *e.g.*, form factors, mobile platforms, and Web browsers. Yet, end-users expect Web applications to behave consistently across these platforms. The X-PERT project aims to detect inconsistencies in the way that Web apps are displayed by these platforms. Dually, FMAP attempts to identify similar elements on Web pages that may be rendered differently on different platforms.

Our work differs from these projects in that it uses inconsistencies in apps to identify problems in the app development frameworks. While our work has primarily targeted APIs used to support application logic, future work on testing mobile apps created using Web-based frameworks (*e.g.*, [1, 3, 7]) can possibly use the techniques from X-PERT and FMAP to identify inconsistencies in the way UI elements are displayed across platforms.

Aside from testing techniques for cross-platform apps, a number of recent projects [20–22, 32, 35, 37] have been developing techniques to test mobile apps. The main goal of these techniques is to devise effective input generation techniques for mobile apps, which is challenging because mobile apps are UI-based and event-driven. Although these projects are not directly related to our work, the input generation methods that they develop can potentially be used to identify inconsistencies in the UIs and UI-handling code of cross-platform apps.

Zhong *et al.* [42] consider the related problem of testing cross-language API mapping relations. Such a relation $\langle f_S, f_T \rangle$ encodes that a method f_S implemented in a library written in a source language implements the same features as the method f_T written for an equivalent library in a target language. Zhong *et al.* also use random differential testing as the strategy to identify inconsistencies in these relations. Their findings are similar to ours, and showcase the difficulty of correctly translating functionality across different platforms and languages.

Random and Differential Testing. There is a rich literature on both random testing and differential testing, and both methods have extensively been used for bug-finding. Fuzz-random testing, for example, feeds random inputs to applications under test. Crashing applications are most likely buggy because they do not handle ill-formed random inputs properly. This method has been used to find bugs in UNIX utilities [34] and GUI-based Windows applications [28]. For object-oriented code, JCrasher [27] generates random unit tests, and uses crashes to identify buggy class implementations. Differential testing, originally introduced by McKeeman [33] has recently found a number of interesting applications in security (*e.g.*, [23, 24, 39]).

Random and differential testing can be usefully combined as is evident from our results. This method has previously been used successfully, for instance, to find bugs in compilers [41]. The authors of Randoop also used this method to test two versions of the Java development kit, finding a number of bugs along the way.

Implementing Cross-platform App Frameworks. As already discussed in Section I, there is significant recent interest in techniques to develop cross-platform mobile apps. For this task, the dominant methods are the use of Web-based frameworks, which support app development in Web-based languages, and native frameworks, which create apps that can natively execute on the platform. These frameworks do much of the leg-work necessary to translate API calls across platforms. To our knowledge, these translations are created manually by domain experts. The software engineering research community has proposed methods to automatically harvest cross-platform API mappings by mining existing code bases (*e.g.*, [29, 38, 43]). Such techniques could potentially be used to improve the way that cross-platform app development frameworks are built.

IX. SUMMARY AND FUTURE WORK

Developers are eager to deploy their mobile apps on as many platforms as possible, and cross-platform mobile app development frameworks are emerging as a popular vehicle to do so. However, the frameworks themselves are complex and difficult to develop. Using X-Checker to test Xamarin, we showed that differential testing is an effective method to identify inconsistencies in the way that these frameworks handle the APIs of the home and target platforms.

While X-Checker has been highly effective, it suffers from a number of limitations that we plan to remedy in future work. First, while X-Checker uses random method sequences as test cases, the arguments to these methods are drawn at random from a fixed, manually-defined pool. We plan to investigate techniques to invoke methods with random, yet meaningful arguments, which would further increase the coverage of the API during testing. Second, X-Checker has primarily focused on testing the framework libraries that provide support for the platform-independent part of cross-platform apps. However, when end-users interact with apps that have been cross-compiled, they also expect a similar end-user experience when interacting with the app’s UI. To ensure this property, we must test that semantically-similar UI elements on different platforms elicit the same behavior within the apps on the corresponding platforms. This will likely require an analysis of the elements of the UI itself, and recent work on cross-platform feature matching [26] may help in this regard. Finally, we plan to extend X-Checker to work with other target platforms (*e.g.*, Xamarin for iOS) and with other cross-platform app-development tools.

X. ACKNOWLEDGMENTS

We thank the reviewers for their comments and Thomas Ball for encouraging us to consider Xamarin as our experimental platform. This research was funded in part by NSF grants 0952128, 1408803 and 1441724.

REFERENCES

- [1] Adobe PhoneGap. <http://phonegap.com>.
- [2] Apportable – Objective-C for Android. www.apportable.com.
- [3] IBM MobileFirst platform foundation. <http://www-03.ibm.com/software/products/en/mobilefirstfoundation>.
- [4] Json.NET – popular high-performance JSON framework for .NET. <http://james.newtonking.com/json>.
- [5] Mono – Cross-platform, open source .NET framework. <http://www.mono-project.com>.
- [6] MyAppConverter – develop once, run anywhere. <http://www.myappconverter.com>.
- [7] Sencha: HTML5 app development. <http://www.sencha.com>.
- [8] Xamarin – Mobile App Development and App Creation Software. <http://xamarin.com>.
- [9] Xamarin Bug 25895. Wrong exception is thrown when calling System.Globalizati.on.CharUnicodeInfo.GetNumericValue with invalid index. https://bugzilla.xamarin.com/show_bug.cgi?id=25895.
- [10] Xamarin Bug 27901. XmlConvert.ToString returns wrong value. https://bugzilla.xamarin.com/show_bug.cgi?id=27901.
- [11] Xamarin Bug 27910. XmlParserContext constructor not throwing XmlException when it should. https://bugzilla.xamarin.com/show_bug.cgi?id=27910.
- [12] Xamarin Bug 27922. XmlConvert.ToUnit throwing wrong/inconsistent exception. https://bugzilla.xamarin.com/show_bug.cgi?id=27922.
- [13] Xamarin Bug 27982. Inconsistent behavior in DynamicAttribute.Equals. https://bugzilla.xamarin.com/show_bug.cgi?id=27982.
- [14] Xamarin Bug 28017. NameTable.Add throwing wrong/inconsistent exception. https://bugzilla.xamarin.com/show_bug.cgi?id=28017.
- [15] Xamarin Bug 28123. Inconsistent behavior in System.Xml.XmlReaderSettings.MaxCharactersInDocument. https://bugzilla.xamarin.com/show_bug.cgi?id=28123.
- [16] Xamarin Bug 28134. System.Text.EncoderFallbackException inconsistent initial state compared to .NET (two inconsistencies assigned to this bug identifier). https://bugzilla.xamarin.com/show_bug.cgi?id=28134.
- [17] Xamarin Bug 28562. Incorrect System.Numerics.Complex.Pow result. https://bugzilla.xamarin.com/show_bug.cgi?id=28562.
- [18] Xamarin Bug 28571. Incorrect behavior in System.Numerics.BigInteger. https://bugzilla.xamarin.com/show_bug.cgi?id=28571.
- [19] Xamarin Bug 28572. Incorrect/inconsistent behavior in System.Numerics.Complex.Divide. https://bugzilla.xamarin.com/show_bug.cgi?id=28572.
- [20] S. Anand, M. Naik, H. Yang, and M. Harrold. Automated concolic testing of smartphone apps. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2012.
- [21] T. Azim and I. Neamtii. Targeted and depth-first exploration for systematic testing of Android apps. In *ACM Symposium on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2013.
- [22] P. Brooks and A. Memon. Automated GUI testing guided by usage profiles. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2007.
- [23] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using Frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *IEEE Symposium on Security and Privacy (Oakland)*, 2014.
- [24] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *USENIX Security Symposium*, 2007.
- [25] S. Choudhary, M. Prasad, and A. Orso. X-PERT: Accurate identification of cross-browser issues in Web applications. In *International Conference on Software Engineering (ICSE)*, 2013.
- [26] S. R. Choudhary, M. R. Prasad, and A. Orso. Cross-platform feature matching for Web applications. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2014.
- [27] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software-Practice and Experience*, 34(11), 2004.
- [28] J. E. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *USENIX Windows Systems Symposium*, 2000.
- [29] A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring likely mappings between APIs. In *International Conference on Software Engineering (ICSE)*, 2013.
- [30] S. Guthrie, T. Myerson, and S. Nadella. Day one keynote presentation, Microsoft Build developer conference, April 2015. <http://channel9.msdn.com/Events/Build/2015/KEY01>.
- [31] H. Heikotter, T. Majchrzak, and H. Kuchen. Cross-platform model-driven development of mobile applications with MD². In *ACM Symposium on Applied Computing (SAC)*, 2013.
- [32] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2013.
- [33] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1), December 1998.
- [34] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM (CACM)*, 33(12), December 1990.
- [35] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon. GUITAR: An innovative tool for automated testing of GUI-driven software. *Journal of Automated Software Engineering*, 21(1), 2014.
- [36] C. Pacheco, S. K. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering (ICSE)*, 2007.
- [37] M. Pradel, P. Schuh, G. Necula, and K. Sen. Event-break: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *ACM Symposium on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2014.
- [38] M. P. Robillard, E. Boddien, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering (TSE)*, 39(5), May 2013.
- [39] V. Srivastava, M. D. Bond, K. S. McKinley, and V. Shmatikov. A security policy oracle: Detecting security holes using multiple API implementation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.
- [40] C. Velazco. Microsoft invites Android and iOS apps to join Windows 10, April 2015. <http://www.engadget.com/2015/04/29/android-ios-apps-on-windows-10>.
- [41] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.
- [42] H. Zhong, S. Thummalapenta, and T. Xie. Exposing behavioral differences in cross-language API mapping relations. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2013.
- [43] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *International Conference on Software Engineering (ICSE)*, 2010.