Towards Self-healing Smartphone Software via Automated Patching

Tanzirul Azim
Univ. of California, Riverside
Riverside, CA
USA
mazim002@cs.ucr.edu

Iulian Neamtiu
Univ. of California, Riverside
Riverside, CA
USA
neamtiu@cs.ucr.edu

Lisa Marvel
US Army Research Laboratory
Aberdeen Proving Ground, MD
USA
lisa.m.marvel.civ@mail.mil

ABSTRACT

Frequent app bugs and low tolerance for loss of functionality create an impetus for self-healing smartphone software. We take a step towards this via on-the-fly error detection and automated patching. Specifically, we add failure detection and recovery to Android by detecting crashes and "sealing off" the crashing part of the app to avoid future crashes. In the detection stage, our system dynamically analyzes app execution to detect certain exceptional situations. In the recovery stage, we use bytecode rewriting to alter app behavior as to avoid such situations in the future. When using our implementation, apps can resume operation (albeit with limited functionality) instead of repeatedly crashing. Our approach does not require access to app source code or any system (e.g., kernel-level) modification. Experiments on several real-world, popular Android apps and bugs show that our approach manages to recover the apps from crashes effectively, timely, and without introducing overhead.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability, Validation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools,Tracing*

Keywords

automatic patch construction; automated software repair; smartphone applications; bytecode rewriting; self-healing software; Google Android

1. INTRODUCTION

As smartphones and tablets continue to increase in popularity [14, 13], more and more critical software (e.g., financial, military, medical apps) shifts to these new platforms. Unfortunately, smartphone software (from the OS to libraries to apps) has a high defect rate [9] due to many factors, including the novelty and rapid evolution pace of smartphone software, the low barrier to entry for publishing software via app marketplaces, as well as the myriad devices and user-specific configurations on which it is running. Maintaining certain functionality (such as the ability to place phone calls) is critical on smartphones, and unlike on desktop systems, we cannot always rely on network connectivity for downloading and applying a patch to fix the bug. Hence there is a strong impetus for self-healing smartphone software. We take a step towards this by presenting an approach for automating patch construction to recover from and prevent future crashes in Android apps.

Our approach is more suitable for smartphone apps than, say, desktop or server programs, due to the compartmentalized nature of smartphone apps: many pieces of functionality, e.g., GUI elements, can be turned off without affecting user experience [17]. Our implementation first employs dynamic analysis to detect when an app has entered an error state and to identify the offending part of the app; then it implements recovery, either by eliminating transient faults and continuing to run at full functionality, or rolling back to a safe state followed by sealing-off the offending part and operating in a limited mode while avoiding further crashes. An example would be a bug in the auto-completion code that crashes the smartphone's Dialer app whenever the user tries to dial a number: when we detect the crash, rather than rendering the whole app inoperable and unable to place emergency (911) calls, we create and apply a patch to turn off auto-completion, hence containing the damage and allowing the Dialer app to continue to run (albeit with some limitations). While this is an incipient form of self healing in smartphone software, it is compelling nevertheless.

Exceptional conditions or bugs have many causes, and manifest in a variety of ways: unhandled exceptions, assertion failures, system overload; our framework detects several such exceptional conditions and reacts accordingly, depending on whether the fault is transient or persistent.

As the context of the error may be different in different situations, it is unreasonable/infeasible to expect the smartphone user to know how to circumvent the error and keep the app functional. Hence our system automatically detects the error and changes the app's behavior to respond to these circumstances so that most of the normal app workflow is not hampered. The approach is centered around several techniques which facilitate self-healing: (1) extracting a high-level model of app operation which captures legal app transitions as a graph; (2) continuous monitoring to detect crashes; (3) identifying and sealing-off the offending app component through app bytecode rewriting.

Although crash prevention has value and is preferable, smartphone software bugs are a fact of life, so in this work we will outline our ideas regarding recovering (potentially lim-

ASE 2014
Copyright 2014 ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642955.

ited) functionality, as follows. Using the statically-constructed model and the crash point, the app goes to a "rollback" point and depending on the nature of the fault (transient or persistent) it creates appropriate conditions, potentially via seal-off, to avoid future crashes.

We have implemented a prototype that equips Android apps with the aforementioned self-healing functionality; we have chosen Android as a target due to its leader status [7]. Our framework is robust, running on actual phones and supporting widely popular apps such as K-9 Mail and Facebook Mobile. Moreover, our approach does not require access to the app source code or modifications to the kernel or libraries; rather we rely on static analysis and rewriting at the bytecode level.

We have evaluated our implementation on a set of bugs in real-world, popular Android apps running on Motorola Droid Bionic phones. Experiments show that our implementation manages to successfully perform self-healing without prohibitive overhead, and the self-healing process is accomplished very efficiently, in less than one minute.

We expect that the fault information revealed by our system could provide feedback to the app developers to help them develop bug-fixing patches.

2. APPROACH

We first present a brief overview of the Android platform, discuss self-healing in the context of the platform and then present our approach in detail.

Android platform. The Android platform consists of the Android OS (a Linux kernel customized for smartphones), the Dalvik VM (virtual machine) and apps, written mostly in Java, that are compiled to a bytecode format named DEX and execute in the Dalvik VM. Apps can also embed native code (written in C/C++) that is executed directly by the OS. Apps have access to a rich set of libraries (Android Application Framework), that offer services from hardware abstractions to orchestrating the GUI control flow.

Self-healing in the context of Android apps. Self-healing computing systems' capabilities include inferring (1) ways of detecting failures (e.g., due to system malfunctions such as exceptions, violations of operational constraints), and (2) strategies for applying corrections to restore (some or all) system functionality. The key concept behind our self-healing mechanism in Android is that an app must resume to a normal GUI state after the app experiences a failure. Hence, it is key that we discover a model (set of GUI states) beforehand, so during recovery the app can be driven to an appropriate state and avoid future crashes. For the purpose of this work, we focus on two types of Android app faults: *transient* and *persistent.* Transient faults occur when operations fail as a result of resource unavailability and will disappear if the operation executes again if the resource becomes available; a simple and effective recovery strategy for these is re-execution. Persistent faults do not go away via re-execution, e.g., because they are due to errors in application logic; in such cases, sealing-off the offending operation is an effective recovery strategy.

Our approach consists of the following phases:

- **Model construction and rollback point identification.** In this phase we identify discrete, safe and unsafe points in the app (which form the basis for our approach), as well as transitions between them, using static analysis.

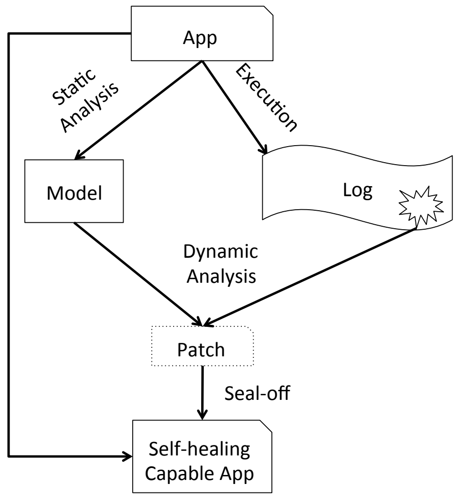

Figure 1: System architecture.

- **Detection.** In this phase, our framework performs dynamic analysis on systems's behavior and app output (e.g., system-wide resource usage, app method calls, GUI elements, privileged actions) to detect crashes and identify faulty components.
- **Recovery.** Our recovery mechanism works in two phases. First, after a crash point is detected, we identify a safe rollback point and if needed (depending on the nature of the fault), we seal off the bytecode associated with the crash point by using the model to identify the faulty part of the binary and then rewrite the bytecode to avoid future execution of code associated with the crash point. Second, we restart the app to a nearby safe point so that users can continue their work and interaction with the app.

2.1 Architecture

Figure 1 shows our system architecture, centered around detecting crashes and in response applying seal-off patches. A *model* is constructed first, via static analysis on the *app* (bytecode); the model includes rollback (resume) points where the app will be driven when recovering from a fault. Next, the app is executed, either manually by users or automatically via systematic exploration tools and its execution *log* is monitored via dynamic analysis. When a failure is detected, we employ bytecode rewriting (code generation and instrumentation) to create a *patch.* We apply the patch via bytecode rewriting; the patch seals off the functionality responsible for the crash, yielding a *self-healing capable app.* We now describe our system's operation in detail.

2.2 Model Construction and Rollback Point Identification

The app model forms the basis for identifying safe and unsafe points in the app. Safe points will be used for rollback and unsafe points will be sealed off. The model, named *Static Activity Transition Graph* (SATG [17]), is a transition graph where nodes are app screens ("activity" = GUI screen in Android parlance) and edges represent possible transitions between screens (which will take place, e.g., as the user navigates around using the GUI). For example, a directed edge from activity A to activity B points to a valid transition from A to B as a result of the user exercising a GUI object associated with A.

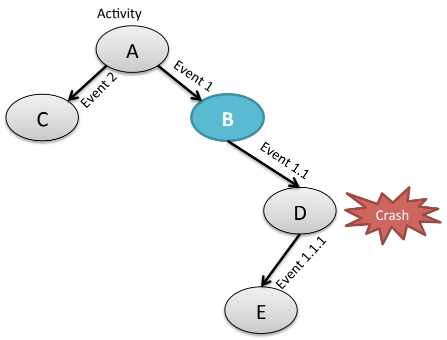

Figure 2: Our static analysis infers B as the rollback point when the app crashes at point D.

Note that SATG construction is not a contribution of this work, but the SATG is nevertheless essential to identifying rollback points and taking recovery actions: in the case of a fault, we calculate the nearest "safe" activity that can be used as a rollback point. Static analysis is important because it reveals the sequence of callbacks associated with activity transitions: invoking these callbacks (which in normal user interaction corresponds to exercising a sequence of GUI elements) allows us to reach the rollback point. Figure 2 shows a SATG constructed with our A^{3}E tool [17]. Here the root node A is the initial activity. Each edge to the next node is labeled with the callbacks or events triggering that transition. For example Event 1 is responsible for the transition from activity A to activity B. Suppose the app crashes at activity D (marked in the figure). From the SATG we can see that activity D was reached via activity B, so activity B is the nearest safe point to restart the app. More generally, rollback points can be obtained via a backward traversal from a crash point.

2.3 Detection

We now discuss *what we detect* —classes of faults in Android app—and *how we detect* them via monitoring.

What we detect. We begin by presenting several common classes of Android apps faults, along with app names that contain these faults (in certain versions). Note that these faults are not particular to Android, as they affect other smartphone platforms as well.

(a) **Resource Shortage/Unavailability.** Unlike the desktop or server platforms, resource availability cannot be taken for granted. For example, smartphone multitasking is much more restrictive: when an app is not in focus it is placed on a stack and essentially put to sleep, its resources taken away and assigned to apps in focus. Apps not properly designed to work with this kind of behavior may experience failure because of resource shortage.

(b) **Unhandled Exceptions.** These failures are mostly due to poor programming practices and inadequate testing that result in failure to anticipate and handle the potential exceptions raised by the app or the system (e.g., NPR News, SoundCloud, K-9 Mail).

(c) **Crashes due to Semantic Errors.** This is a broad class of errors; for example the app fails to accepts certain types of input that otherwise should be accepted and dealt with by the program. For example, an app crashes because the input file is not in the correct format or broken, hence the app crashes instead of generating appropriate warnings (APV PDF Viewer).

(d) **Crashes due to Loss of Network Connectivity.** Most Android apps communicate with remote servers. Even the apps which do not require a network to carry on their functionality may still require network access for loading advertisements. However, Internet connectivity might be intermittent, hence apps must deal with situations where network access is temporarily unavailable (e.g., Facebook Mobile).

(e) **Permission Violations.** In Android, access to sensitive resources is protected by a set of permissions. When the app tries to access resources or functionality it does not have permission for, the OS will terminate the app.

(f) **IPC errors.** Inter Process Communication (IPC) is heavily used in Android for isolation and security. Apps must abide by the IPC communication protocol; failure to do so may lead to apps being terminated.

How we detect. Currently our detection strategy relies on Android's system-wide logging facility (`logcat`). In Android, the Dalvik VM constantly monitors the app and when a fault is encountered, the VM reports the potential cause of error and the associated methods or callback into the `logcat`. To implement monitoring, we add a listener in the Dalvik VM's logging system and in the event of a fault, we isolate the exact method and activity (screen) responsible for the fault.

2.4 Recovery

Example: recovering from a bug in NPR News. We first illustrate how our system recovers from an actual bug in the NPR News app (Figure 3). App execution starts from the root activity, NewsListActivity. The A^{3}E systematic explorer clicks a menu button to get the hourly news update, which takes the app to HourlyNewsActivity. Then A^{3}E plays the radio stream repeatedly. This initiates a service component, PlaybackService. At this time the program enters an illegal state and crashes; the crash is captured in the log. Analyzing the log, our system finds that the closest method associated with the crash is prepareThenPlay in the service class PlaybackService. This concludes the online fault detection phase. Next, in the recovery phase, we apply a seal-off patch to prepareThenPlay, as described next.

Constructing seal-off patches. We sketch the construction of the bytecode patch (inserting code in the app via binary rewriting, achieving seal-off) in Figure 4. Suppose erroneousMethod is the method associated with the fault. First, we surround the original method code with a generic exception handler. Upon failure, the handler will just return (because the original method's return type is **void**), thus preventing executing the erroneous code. Custom code can be added at this point, e.g., to perform more extensive checkpointing. In general, though, the returned value will have to be of the same type as the original mehtod's return type, hence we create a return object of the appropriate type.

Next, for methods containing activities (recall that an activity roughly corresponds to a screen in Android), we apply a similar technique, inserting a try/catch block around the onCreate virtual method from the Activity class. The onCreate method is called when the activity is loaded (i.e., the screen is displayed). If the sealed-off method execution generates further exceptions, the handler will catch the exception and refresh the activity. Thus the activity will remain operational. With the above technique, we gain two

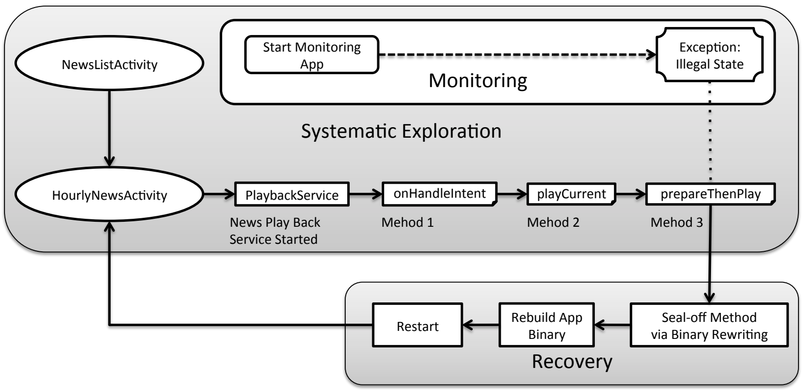

Figure 3: Example: fault point detection and rollback in the NPR News app.

```
//inside the faulty method
void erroneousMethod(T param)
{
   //surround method with try/catch block
   try
   {
      // original method code
   }
   catch(Exception e) //generic exception handler
   {
      //write custom exception handling code
      return;
   }
}

//inside the activity
@Override
protected void onCreate(Bundle savedInstanceState)
{
   //Surround with generic try−catch block
   try
   {
      // initialize activity and load GUI components
   }
   catch(Exception e)
   {
      //write custom error handling code
      //refresh the activity
       startActivity (getIntent ());
   }
}
```

Figure 4: Code sketch of patch construction.

advantages: first, by sealing off just the actual problematic method we are ensuring the least amount of functionality loss; second, we are limiting the functionality seal-off only in the time of an actual fault—the rest of the time the app will behave normally. We thus implement a demand-driven approach, with self-healing taking over only when necessary, and minimizing operational limitations.

The general technique. Our failure detection is dynamic hence it takes an actual execution to find and recover from a crash. When the system is used "in the wild," users interact with the app as they normally do, and if the app crashes, users will experience a small delay due to recovery. For this paper, however, we used an automated exploration tool we develop in prior work, $\mathsf{A^3E}$ [17] to drive the execution, so we could reliably drive the app into a state where it crashes. In the background, we constantly monitor the VM log for events that may indicate failure (Section 2.3).

When a failure does occur, we determine the finest granularity level for inserting our fault-avoiding code. Note that we have several options here. First, we could mark the entire current activity as the potential fault point and deny access to the activity; but this is not realistic, as an activity contains many other GUI features that may be completely unrelated to the fault observed. Second, we could limit the functionality of the associated GUI object. For example, if the crashing GUI object is a button we can disable it. But this may be also unrealistic. For example, for some inputs the button code may fail, but it will work on other inputs. Third, we can operate (seal off) at the method level. Therefore, the method is the finest seal-off granularity; we employ this granularity level in our approach by assigning the fault to the crashing method. For example, in Figure 3 we will seal-off the third method, prepareThenPlay (using the patching procedure explained above) because it is on the lowest level of the exception trace.

Once a crash point is reached, we rollback and resume the app. The rollback point depends on whether the crash is transient or persistent.

For *transient errors* (generated in response to external events such as illegal sensor data, unexpected shared memory deletion by the Android OS, background services shutdown due to low energy, network unavailability, resource shortage, etc.) the rollback point is the point of the crash. The idea is that after rollback and restart these transient environmental exceptions may not be raised and the app can resume its functionality normally.

For *persistent errors* (e.g., unhandled exceptions, semantic errors, IPC communication errors, unauthorized access), we rollback to an earlier point (previous node in the SATG) and use bytecode rewriting to seal off the faulty method in the faulty SATG node. While this limits functionality, it ensures that the app will not call the offending method again.

3. IMPLEMENTATION

Platform. We implemented our approach and conducted experiments on a Motorola Droid Bionic phone, running Android 2.3.4 (note, however, that the test results can also be achieved by running the app in the emulator).

Tools. For model construction we used the SATG extraction component (static analysis-based) of $\mathsf{A^3E}$. To drive exploration, we used the systematic exploration component of $\mathsf{A^3E}$. Bytecode rewriting was done using the smali Dalvik assembler/disassembler [1]. We wrote the main instrumentation code in Java.

In our current setup the phone was "tethered" to a laptop; this was necessary for running $\mathsf{A^3E}$, smali, and initiating rollback/restart. However, we expect that in the future the approach will run solely on the phone, as we envision it should run "in the wild," with no tethering required.

App	Version	Bug Type	Size	
			Kinst.	KBytes
Facebook Mobile	1.6.0	Network Unavailability	173	3,000
NPR News	2.1b	Semantic Error	21	70
K-9 Mail	4.0.0.3	Unhandled Exception	157	2,300
SoundCloud	1.2.2	Unhandled Exception	48	250
APV PDF Viewer	0.2.7	Semantic Error	3	1,100

Table 1: Examined apps.

App	Static analysis	Systematic exploration (time-to-crash)	Self-healing	
			Bytecode rewriting	Rollback and restart
	(seconds)	(seconds)	(seconds)	(seconds)
Facebook Mobile	86	4	0	9
NPR News	53	22	22	9
K-9 Mail	94	52	44	6
SoundCloud	51	16	17	4
APV PDF Viewer	34	7	13	4

Table 2: Efficiency measurements results.

4. EVALUATION

Examined apps. For evaluation we chose several sizable, popular Android apps that contained known bugs. In Table 1 we present the apps: version, type of bug, and app size. According to Google Play, each app was highly popular, with more than 1 million downloads. We have evaluated our approach in terms of *effectiveness*, i.e., can the system recover from actual bugs in popular sizable apps? and *efficiency*, i.e., is the overhead of our approach acceptable?

Effectiveness. Our approach was effective at performing self-healing in response to three categories of bugs encountered in five popular apps.

Efficiency. Our approach incurs a one-time overhead for model extraction, via static analysis, to enable rollback point detection. The second column of Table 2 shows the static analysis time for each app. Model extraction time is solely depending on the app's binary size and code complexity, and as it is a one-time cost incurred *before running the app*, we believe that the 34–94 seconds figure is acceptable.

We drove the apps to crash points via systematic exploration. Depending on the bug, exploration time will vary, though techniques such as targeted exploration [17] or fast-forwarding record-and-replay [10] can significantly accelerate the procedure. The time-to-crash is presented in the third column of Table 2. For example, for Facebook Mobile, the actual fault was in the initial login screen, hence the systematic exploration time (4 seconds) was much lower than for the other apps.

As mentioned in Section 2.4, self-healing might require bytecode rewriting (if seal-off is involved) and always requires rollback and restart. The bytecode rewriting time (performed only once after the crash, for non-transient bugs) depends on the size of the app. This time is shown in the fourth column of Table 2: 13–44 seconds. Facebook Mobile required no rewriting because it experiences a transient bug, hence the '0' figure for rewriting time. Finally, the time required for rollback and restart is shown in the last column of Table 2. The rollback time involves uninstalling the current version, installing the modified app, and rolling back to the nearest safe point within the app. While just rolling back requires very little time (in our case not more than 1 second), uninstalling the current faulty app and reinstalling the modified app takes longer, 3–8 seconds. However this is much shorter than any manual rollback and restart because not only a human would require longer time to uninstall and reinstall but also a human would restart the app from the home screen and therefore would take longer to reach the former point (the point where the app was before the crash). As we rollback to the nearest safe point, we can ensure faster exploration to the safe state. As shown in the last column of Table 2, our automated rollback required at most 9 seconds for the apps. Hence the total self-healing time is 9–50 seconds, which we believe is acceptable.

For transient faults, recovery is faster because we do not rewrite the app: a simple rollback is usually enough to resume normal behavior. For example, our examined version of Facebook Mobile failed when there was no network connectivity. A rollback restored the app and as the connectivity was reestablished, the app resumed its normal operation. Hence recovery was faster than for the other apps, as no bytecode rewriting was performed. Note that app performance is not affected by seal-off, since only a specific part of a method's bytecode (i.e., the prologue) is rewritten.

4.1 Limitations

Our prototype is subject to several limitations that we intend to address in future work.

First, mobile apps tend to be GUI-centric, so upon rollback and restart we only lose GUI state such as previously-entered data, or selected items. For more stateful scenarios, we will have to perform more sophisticated healing operations (e.g., more sophisticated fault detection analyses and more extensive checkpoint and rollback).

Second, our approach is *reactive* (responds to bugs after they manifest), rather than *proactive*. We expect that, using techniques such as consistency constraints or invariant checking, we can detect and fix errors before they develop into full-fledged crashes.

Third, in our current implementation, the phone was tethered to a laptop. There is however no fundamental hurdle to running the approach entirely on the phone. We used tethering for systematic exploration (which will not be necessary when apps crash "in the wild"); and to benefit from existing app rewriting support offered by desktop tools.

5. RELATED WORK

Self-healing and automated patch construction have been studied in many contexts, from clusters of Internet servers [16, 2] to web browsers [12]. Demsky et al. [6] use formal specifications for data structures that allow integrity properties in data structures to be monitored and data structures to be repaired in case the specification is violated. Perkins et al. [12] introduced a system named ClearView that monitors an application's execution to learn application invariants, detect failures, and in case of failure automatically constructs and applies a patch to heal the application. ClearView has been applied to Firefox with a high degree of success and resilience to attacks. Sidiroglou et al. [15] developed an approach named ASSURE that employs rescue points to recover from unanticipated failures in desktop/server Linux applications. ASSURE utilizes online code injection and restores program execution to a rescue point where existing error handling mechanism is used to inject fault recovery code. Candea et al. [3] have proposed "microreboots" (rebooting small components instead of entire applications) as a recovery technique for Internet services. Sultan et al. [16] and Bohra et al. [2] use remote DMA to perform peer monitoring and take-over in a cluster. to provide seamless service to clients. However, to the best of our knowledge, we are the first to study self-healing on the smartphone platform.

Wei et al. [18] proposed an automated patch generation technique based on contracts. Their approach is limited to systems built using the design-by-contract pattern. Although their strategy has shown promising results, smartphone apps are not developed using design-by-contract.

Weimer et al. [19] demonstrated a fully automated, genetic programming approach for finding and fixing bugs. Their tool, *GenProg*, identifies legal program variants for positive test cases and they generate fixes with the means of structural differences and delta debugging upon the correct program variant for the faulty input. Michail et al. [11] proposed a scheme to use user-generated bug reports to predict future bugs in a software execution path to warn the users to avoid that path. Their scheme is based on predicting the presence of faults in a particular execution based on previous reports from the users. The work of Kim et al. [8] generates automatic patches from already existing patches written by human developers. They manually inspected the human written patches and automatically develop the repair code by identifying common fix patterns. Their approach requires manual effort and might not be always practical to employ in a quick succession which is required in mobile platforms with shorter update cycle. In contrast to these three efforts, our approach does not rely on test cases or bug reports, but rather reacts dynamically to a set of predefined errors.

Dallmeier et al.'s approach [5] automatically extracts anomalies in object behavior and generate patches accordingly. This idea may be useful on smartphone apps, but it does not guarantee sealing-off faulty code in a deterministic manner, e.g., random events can occur in a particular executions and the same set of faults can manifest differently.

Carzaniga et al. [4] employ code rewriting to work around API-related faults in web applications. They have showed their approach in popular web APIs such as Google maps and YouTube. While the are similarities (e.g., event-driven) between smartphone and web apps, there are significant differences: smartphone apps are centered around rich gestures and sensors, so it is unclear how their approach would translate to smartphones.

6. CONCLUSIONS

We have presented an approach that uses automatic error detection and patch construction towards providing a certain degree of self-healing capabilities to Android apps. We use dynamic analysis to identify crash points, static analysis to identify rollback points, and binary rewriting to seal off methods associated with crash points so that apps can continue to function even after a crash, albeit with limited functionality. Through experiments on actual bugs in several popular apps, we show that our approach is effective and reasonably efficient.

Acknowledgments

We thank the anonymous referees for their helpful comments. This research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. This research was also supported in part by NSF grant CNS-1064646.

7. REFERENCES

[1] Smali: An assembler/disassembler for Android's dex format. `http://code.google.com/p/smali/`.

[2] A. Bohra, I. Neamtiu, P. Gallard, F. Sultan, and L. Iftode. In *ICAC'04*, pages 256–263.

[3] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot: A technique for cheap recovery. In *OSDI'04*.

[4] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *FSE '10*, pages 237–246.

[5] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *ASE '09*.

[6] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *ICSE'05*.

[7] IDC. Android Pushes Past 80% Market Share While Windows Phone Shipments Leap 156.0% Year Over Year in the Third Quarter, Novemeber 2013. `http://www.idc.com/getdoc.jsp?containerId=prUS24442013`.

[8] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE '13*, pages 802–811.

[9] A. Kumar Maji, K. Hao, S. Sultana, and S. Bagchi. Characterizing failures in mobile oses: A case study with android and symbian. In *ISSRE'10*.

[10] L. Gomez, I. Neamtiu, T.Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *ICSE '13*.

[11] A. Michail and T. Xie. Helping users avoid bugs in gui applications. In *ICSE '05*, pages 107–116.

[12] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP '09*.

[13] Pew Research Center. Report: Mobile Tablet Ownership 2013. `http://pewinternet.org/Reports/2013/Tablet-Ownership-2013.aspx`.

[14] Pew Research Center. Report: Smartphone Ownership 2013. `http://pewinternet.org/Reports/2013/Smartphone-Ownership-2013.aspx`.

[15] S. Sidiroglou, O. Laadan, C. R. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: Automatic software self-healing using rescue points. In *ASPLOS'09*.

[16] F. Sultan, A. Bohra, S. Smaldone, Y. Pan, P. Gallard, I. Neamtiu, and L. Iftode. Recovering internet service sessions from operating system failures. *Internet Computing, IEEE*, 9(2):17–27, 2005.

[17] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *OOPSLA '13*.

[18] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *ISSTA '10*, pages 61–72.

[19] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE '09*, pages 364–374.