

Automating Software Development for Mobile Computing Platforms

Kevin Patrick Moran

Maitland, FL

Bachelor of Arts, College of the Holy Cross, 2013
Master of Science, College of William & Mary, 2015

A Dissertation presented to the Graduate Faculty
of the College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
May 2018

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

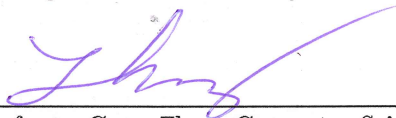


Kevin Patrick Moran

Approved by the Committee, May 2018



Committee Chair
Associate Professor Denys Poshyvanyk, Computer Science
College of William & Mary



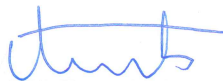
Associate Professor Gang Zhou, Computer Science
College of William & Mary



Assistant Professor Xu Liu, Computer Science
College of William & Mary



Assistant Professor Adwait Nadkarni, Computer Science
College of William & Mary



Professor Andrian Marcus, Computer Science
The University of Texas at Dallas

COMPLIANCE PAGE

Research approved by

Protection of Human Subjects Committee

Protocol number(s): PHSC-2015-08-11-10536-dposhyvanyk
PHSC-2014-06-16-12172-dposhyvanyk

Date(s) of approval: 08/19/2015
07/08/2017

ABSTRACT

Mobile devices such as smartphones and tablets have become ubiquitous in today’s computing landscape. These devices have ushered in entirely new populations of users, and mobile operating systems are now outpacing more traditional “desktop” systems in terms of market share. The applications that run on these mobile devices (often referred to as “apps”) have become a primary means of computing for millions of users and, as such, have garnered immense developer interest. These apps allow for unique, personal software experiences through touch-based UIs and a complex assortment of sensors. However, designing and implementing high quality mobile apps can be a difficult process. This is primarily due to challenges unique to mobile development including change-prone APIs and platform fragmentation, just to name a few.

In this dissertation we develop techniques that aid developers in overcoming these challenges by automating and improving current software design and testing practices for mobile apps. More specifically, we first introduce a technique, called `GVT`, that improves the quality of graphical user interfaces (GUIs) for mobile apps by automatically detecting instances where a GUI was not implemented to its intended specifications. `GVT` does this by constructing hierarchal models of mobile GUIs from metadata associated with both graphical mock-ups (*i.e.*, created by designers using photo-editing software) and running instances of the GUI from the corresponding implementation. Second, we develop an approach that completely automates prototyping of GUIs for mobile apps. This approach, called `REDRAW`, is able to transform an image of a mobile app GUI into runnable code by detecting discrete GUI-components using computer vision techniques, classifying these components into proper functional categories (*e.g.*, button, dropdown menu) using a Convolutional Neural Network (CNN), and assembling these components into realistic code. Finally, we design a novel approach for automated testing of mobile apps, called `CRASHSCOPE`, that explores a given Android app using systematic input generation with the intrinsic goal of triggering crashes. The GUI-based input generation engine is driven by a combination of static and dynamic analyses that create a model of an app’s GUI and targets common, empirically derived root causes of crashes in Android apps.

We illustrate that the techniques presented in this dissertation represent significant advancements in mobile development processes through a series of empirical investigations, user studies, and industrial case studies that demonstrate the effectiveness of these approaches and the benefit they provide developers.

TABLE OF CONTENTS

Acknowledgments	viii
Dedication	ix
List of Tables	x
List of Figures	xi
1 Introduction	2
1.1 Overview	2
1.2 Motivation - Language Dichotomies in Software Engineering	4
1.3 Research Context: Mobile Applications	7
1.4 Contributions & Outline	8
2 Background & Related Work	11
2.1 A Brief Introduction to Mobile Software Development	11
2.1.0.1 Unique Aspects and Challenges of the Mobile Development Process	12
2.1.1 Android Development Tools and Frameworks	15
2.1.1.1 Virtual Android Devices	16
2.1.1.2 Android Debugging Bridge	16
2.1.1.3 UI Automator Framework	16
2.1.1.4 Android Logcat	17

2.2	Fundamentals of Mobile Graphical User Interfaces	17
2.2.0.1	GUI-Components, GUI-Containers, and Screens	17
2.2.0.2	Design Violations & Presentation Failures	20
2.3	Work Related to Detection of GUI Design Violations in Mobile Apps	22
2.3.1	Detecting Presentation Failures in Web Applications	22
2.3.2	Cross-Browser Testing	23
2.3.3	Other Approaches for GUI Verification	23
2.4	Work Related to Automated Prototyping of Graphical User Interfaces	
	for Mobile Apps	24
2.4.1	Reverse Engineering Mobile User Interfaces:	24
2.4.2	Mobile GUI Datasets	25
2.4.3	Other GUI-Design and Reverse Engineering Tools:	26
2.4.4	Image Classification using CNNs:	28
2.5	Work Related to Automated Mobile Testing	29
2.5.1	Automation APIs/Frameworks	29
2.5.2	Record and Replay Tools	33
2.5.3	Automated Test Input Generation Techniques	35
2.5.3.1	Random/Fuzz Testing	37
2.5.3.2	Ripping/Systematic Exploration	37
2.5.3.3	Model-Based Testing	38
2.5.3.4	Other Types of Input Generation Approaches	39
2.5.4	Bug and Error Reporting/Monitoring Tools	41
2.5.5	Mobile Testing Services	43
2.5.5.1	CrowdSourced Functional Testing	44
2.5.5.2	Usability Testing	44
2.5.5.3	Security Testing	45
2.5.5.4	Localization Testing	45

2.5.6	Cloud-Based Testing Frameworks	45
2.5.7	Device Streaming Tools	46
3	Automated Reporting of Mobile GUI Design Issues	48
3.1	Problem Statement & Origin	52
3.1.1	Problem Statement	52
3.1.1.1	Problem Statement	52
3.1.2	Industrial Problem Origins	52
3.2	Design Violations in Practice	53
3.2.1	Study Setting & Methodology	54
3.2.2	Grounded Theory Study Results	55
3.3	The GvT Approach	56
3.3.1	Approach Overview	56
3.3.2	Stage 1: Mock-Up GUI Collection	57
3.3.2.1	Mock-Up GUI Collection	57
3.3.2.2	Dynamic App GUI-Collection	58
3.3.3	Stage 2: GUI Comprehension	59
3.3.4	Stage 3: Design Violation Detection	60
3.3.4.1	Perceptual Image Differencing	60
3.3.4.2	Detecting Layout Violations	61
3.3.4.3	Detecting Text Violations	62
3.3.4.4	Detecting Resource Violations	62
3.3.4.5	Generating Violation Reports	63
3.3.5	Industrial Collaboration Methodology	64
3.4	Design of the Experiments	64
3.4.1	Study 1: GvT Effectiveness & Performance	65
3.4.1.1	Study Context	65

3.4.1.2	Synthetic <i>DV</i> Injection	66
3.4.1.3	Study Methodology	67
3.4.2	Study 2: GVT Utility	68
3.4.2.1	Study Context	68
3.4.2.2	Study Methodology	68
3.4.3	Study 3: Industrial Applicability of GVT	69
3.4.3.1	Study Context & Methodology	70
3.5	Empirical Results	70
3.5.1	Study 1 Results: GVT Performance	70
3.5.2	Study 2 Results: GVT Utility	71
3.5.3	Study 3 Results: Industrial Applicability	73
3.6	Limitations & Threats to Validity	75
3.7	Conclusion & Future Work	76
4	Machine Learning-Based Prototyping of GUIs	77
4.1	Background & Problem Statement	83
4.1.1	Convolutional Neural Network (CNN) Background	83
4.1.2	Problem Definition	85
4.2	Approach Description	86
4.2.1	Phase 1 - Detection of GUI-Components	87
4.2.1.1	Parsing Data from Design Mockups	87
4.2.1.2	Using CV Techniques for GUI-component Detection:	89
4.2.1.3	ReDraw Implementation - GUI Component Detection	89
4.2.2	Phase 2 - GUI-component Classification	90
4.2.2.1	Phase 2.1 - Large-Scale Software Repository Mining	
	and Dynamic Analysis	90

4.2.2.2	ReDraw Implementation - Software Repository Mining	
	and Automated Dynamic Analysis	92
4.2.2.3	Phase 2.2 - CNN Classification of GUI-Components	95
4.2.2.4	ReDraw Implementation - CNN Classifier	95
4.2.3	Phase 3 - Application Assembly	104
4.2.3.1	Deriving GUI-Hierarchies	104
4.2.3.2	Inferring Styles and Assembling a Target App	106
4.2.3.3	ReDraw Implementation - App Assembly	107
4.3	Empirical Study Design	109
4.3.1	RQ ₁ : Effectiveness of the CNN	110
4.3.2	RQ ₂ : GUI Hierarchy Construction	112
4.3.3	RQ ₃ : Visual Similarity	115
4.3.4	RQ ₄ : Industrial Applicability	115
4.4	Experimental Results	116
4.4.1	RQ ₁ Results: Effectiveness of the CNN	116
4.4.2	RQ ₂ Results: Hierarchy Construction	118
4.4.3	RQ ₃ Results: Visual Similarity	120
4.4.4	RQ ₄ Results: Industrial Applicability	124
4.4.4.1	Front End Android Developer @Google	124
4.4.4.2	Mobile UI/UX Designer @Huawei	125
4.4.4.3	Mobile Researcher @Facebook	125
4.5	Limitations & Threats to Validity	126
4.5.1	Limitations and Avenues for Future Work	126
4.5.2	Internal Validity	127
4.5.3	Construct Validity	130
4.5.4	External Validity	131
4.6	Conclusion & Future Work	131

5	Improving Automated Android Testing	133
5.1	Background & Motivation	136
5.1.1	Previous Studies on Mobile App Bug/Crashes	136
5.1.2	Limitations of Mobile Testing Approaches	137
5.2	CrashScope Design	138
5.2.1	Extracting Activity and App-Level Contextual Features	140
5.2.2	Exploration of Apps & Crash Detection	141
5.2.3	Testing Apps in Different Contextual States	142
5.2.4	Multiple Execution Strategies	143
5.2.5	Generating Expressive, Natural Language Crash Reports	143
5.2.6	Generating & Replaying Reproduction Scripts	144
5.3	Empirical Study 1: Crash Detection Capability	146
5.3.1	Methodology	147
5.3.2	Results & Discussion	149
5.4	Study 2: Reproducibility & Readability	152
5.4.1	Methodology	153
5.4.2	Results & Discussion	154
5.5	Limitations & Threats to Validity	155
5.6	Conclusions	156
6	Conclusions & Future Research	158
6.1	Toward Automatically Documenting Graphical User Interfaces	159
6.2	Toward GUI-centric Automated Program Understanding & Synthesis	160
6.3	Toward a Practical, Comprehensive Framework for Automated GUI-	
	based Testing	161
6.3.1	The CEL Testing Principles	162
6.4	Concluding Remarks	169

A Individual Contributions to Projects	171
A.1 Individual Contributions to the GVT Project	171
A.2 Individual Contributions to the REDRAW Project	172
A.3 Individual Contributions to the CRASHSCOPE Project	174
A.4 Individual Contributions to the Formulation of	
CEL Mobile Testing	175

ACKNOWLEDGMENTS

This Dissertation and the work performed towards its writing would not have been possible without the support and guidance of several individuals. First and foremost, I would like to thank my advisor, Dr. Denys Poshyvanyk for taking the time to mentor and support a physics major with an initially limited background in computer science. He has tirelessly supported this work and set a great example illustrating how to conduct meaningful research in the field of software engineering with integrity, vision, passion and thoroughness. It goes without saying that without him this work would not have been possible, and I look forward to continuing to grow as a researcher under his mentorship as I pursue my postdoc. I would also like to thank my committee members, Gang, Xu, Adwait, and Andrian for their mentorship and feedback throughout the writing of this Dissertation.

Next, I would like to thank all of the members of the SEMERU group at William & Mary for their support in the projects described in this dissertation and for their mentorship and friendship over my career as a doctoral student. In particular, I would thank to Carlos Bernal-Cárdenas, Mario Linares-Vásquez, Christopher Vendome, Michele Tufano, Michael Curcio, and Dan Jelf. Carlos, your coding expertise and patience in teaching me new techniques has been invaluable. Mario, your research expertise and constructive feedback contributed immensely to my education as a software engineer and research practitioner. Chris and Michele, I am grateful for your support and friendship throughout my time at William & Mary. I cannot possibly give you all enough credit. Michael and Dan, it was a pleasure to advise you as undergraduates and I cannot wait to hear of your future accomplishments.

I thank my parents Brian and Elizabeth, and my brothers Brain, Charles, and Jack for their continued support of my academic pursuits, especially during difficult times. Mom and Dad, I look to you both for inspiration and motivation in the way you have raised my brothers and me and in the way you live your lives. I would also like to thank my grandparents, Tom and Jeanne Moran, and Mary and Paul Lavin, for always encouraging me in my studies and offering crucial advice when needed.

Most importantly, I would like to thank my wife Emily for her unwavering support throughout my doctoral studies. This dissertation would not have been possible without your companionship, advice, and love in both bad and good times. My full appreciation for having you in my life can not be adequately put into words.

To my wife Emily – Your compassion and strength of character have helped me to become a better teacher, researcher, and person. My love for you grows deeper with each passing day.

LIST OF TABLES

2.1 This Table Surveys the current state of tools, frameworks, and services	
that support activities related to mobile testing, originating from both	
Academic and Industrial backgrounds.	30
4.1 Labeled GUI-Component Image Datasets	111
4.2 Semi-Structured Interview Questions for Developers & Designers	113
4.3 Confusion Matrix for REDRAW	116
4.4 Confusion Matrix for BOVW Baseline	117
4.5 Pixel-based comparison by MAE: Mann-Whitney test (p -value) and	
Cliff's Delta (d).	121
4.6 Pixel-based comparison by MSE: Mann-Whitney test (p -value) and	
Cliff's Delta (d).	121
5.1 Tools used in the comparative fault finding study	146
5.2 Unique Crashes Discovered with Instr. Crashes in parentheses	149
5.3 User Experience Results: This table reports the mean average	
response from 16 users regarding the User Experience questions posed	
for both CrashScope generated reports and the original human written	
reports found in the app's issue trackers. (CS = CrashScope Bug	
Reports, O=Original Bug Reports, M =Mean, SD =Standard Deviation)	153

LIST OF FIGURES

1.1 The Hierarchy of Abstraction in Computer Science	3
2.1 Illustration of the GUI Structure of the Pandora Android Application	18
2.2 Examples of Formal Definitions	21
3.1 Distribution of Different Types of Industrial <i>DVs</i>	55
3.2 Overview of GVT Workflow	57
3.3 Study 1 - Detection Precision (<i>DP</i>), Classification Precision (<i>CP</i>), Recall(<i>R</i>)	70
3.4 Study 2 - Developer CP, DP, and R	71
3.5 Study 2 - UX Question Responses. SD=Strongly Disagree, D=Disagree, N=Neutral, A=Agree, SA=Strongly Agree	71
3.6 Study 3 - Applicability Questions. SD=Strongly Disagree, D=Disagree, N=Neutral, A=Agree, SA=Strongly Agree	73
4.1 Typical Components of CNN Architecture	84
4.2 Overview of Proposed Approach for Automated GUI-Prototyping	86
4.3 Heat-map of GUI Components by Category	94
4.4 Example of a subset of ReDraw’s training data set consisting of GUI- Component sub-images and domain (Android) specific labels. Images and corresponding Labels are grouped according to the dashed-lines. . . .	97
4.5 Heat-map of GUI Components by Category After Filtering	98
4.6 Screenshots of synthetically generated applications containing toggle buttons and switches	99

4.7 REDRAW CNN Architecture	102
4.8 Illustration of KNN Hierarchy Construction	105
4.9 Hierarchy similarities based on edit distances	118
4.10 Pixel-based mean average error and mean squared error of screenshots: REDRAW, REMAUI, and pix2code apps	120
4.11 Examples of apps generated with REDRAW exhibiting high visual and structural similarity to target apps	122
5.1 Overview of CrashScope Workflow	138
5.2 Crash Screen-Flow	144
5.3 Example of Contextual Information and Reproduction Steps sections in a generated crash report	145
5.4 Average Coverage Results for the Comparative Study	150

Automating Software Development for Mobile Computing Platforms

Chapter 1

Introduction

1.1 Overview

“The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. This essence is abstract in that such a conceptual construct is the same under many different representations. It is nonetheless highly precise and richly detailed. I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation.”

– Fredrick Brooks, *No Silver Bullet – Essence and Accident in Software Engineering* (1986)

Software developers inherently reason about different abstractions of ideas. In fact, the foundations of computer science more broadly are centered upon a hierarchy of abstractions (Fig. [1.1](#)). This hierarchy begins at the lowest level with the physical representation of computers as a complex assortment of electrical signals, moves toward representations of ideas in code that are able to carry out logical processes, and culminates at the highest level in mental models of programs for solving problems. Thus it can be observed that, at its core, computer science is largely concerned with the interplay between the various levels of this abstraction hierarchy. In his widely regarded “*No Silver Bullet*” essay [\[133\]](#) Fredrick Brooks acknowledges two most common abstractions from this hierarchy that modern software engineers must reason between: (i) *conceptual constructs* (*i.e.*, mental models of a

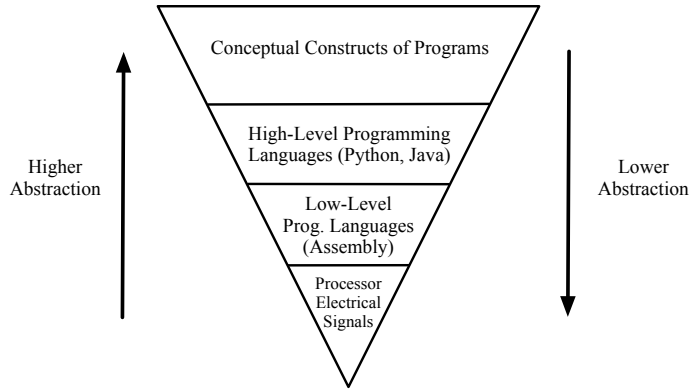


Figure 1.1: The Hierarchy of Abstraction in Computer Science

given software program), and (ii) *representations* of these conceptual constructs (*i.e.*, their concrete instantiation in a medium such as code). Brooks argues that the most critical part of the software development process is the conceptualization of interlocking constructs that inherently constitute a piece of software, not the transferral of these concepts into a concrete representation in code. Although translating abstract mental models into a tangible artifact like code is not a trivial process, Brooks recognized that the mental formulation of *what* needs to be built is the most crucial step, as it is a distinctly abstract process requiring intellectual acuity. This is logically evident, as a faithful instantiation of ineffective ideas in code inevitably results in an unsuccessful program. This conceptualization of the mental model of a program is ultimately what Brooks refers to as the *essence* of software engineering.

With this view of the software development process in mind, there are evidently two major courses of research for improving engineering practices. Namely, conducting studies to understand, and designing techniques to aid and automate: (i) the derivation of a conceptual model embodying the requirements, specifications, and design of a software system, or (ii) the process of translating this conceptual model into a concrete representation that can be understood, executed, and maintained by both humans and computers. However, directly addressing this first course of research is exceedingly difficult, as it attempts to directly operate upon the *essence* of software engineering. As Brooks argues, due to the widely variable nature of software projects and the distinctly unique thought processes of humans, there is unlikely a “silver bullet” that dramatically improves the process of *conceptualizing* software.

Humans are likely to play a major role in the process of software development for the foreseeable future, and as such the process of developing conceptual constructs of software is likely to persist. Methods for reasoning about software at a conceptual level tend to assume a multitude of different forms across varying development domains and teams, and require an high level of ingenuity inherent in most skilled software engineers. Thus, as Brooks suggests, it is difficult to develop any singular notable advancement that aids in this conceptualization process. If this is the case, then how can software engineering researchers help to push the field forward? In this dissertation, we posit that helping to make the instantiation of conceptual software constructs as frictionless as possible can dramatically improve the overall development process. The quicker and easier it is for an engineer to move from concept to code, the faster that ideas can be proved out, and judgements made about conceived programs, which we assert will lead to more effective and efficient creation and maintenance of software. In fact, this direction of work directly targets Brook's prescriptions for dealing with the difficulties that arise related to the *essence* of software engineering namely, *rapid prototyping and iteration* and *growing software organically* [133].

The work presented in this dissertation attempts to facilitate the process of instantiating conceptual software development concepts into accurate, effective representations through automation. The hope is that by automating different parts of the software design, development and testing processes, we will be able to allow developers to focus more effectively on the important task of conceptualizing the data, algorithms and functions that underlie the problem or task to which the software will be applied; thus facilitating the rapid iteration and organic evolution of intuitive, elegant programs.

1.2 Motivation - Language Dichotomies in Software Engineering

As explained in the previous section, the work conducted in the presented dissertation is aimed at designing techniques to *automate* various parts of the software development

process. This automation is meant to facilitate the instantiation of conceptual constructs of software into concrete representations. However, these concepts can be concretely represented in several different manners, such as code, natural language, or in graphical user interfaces (GUIs). In this section, we further motivate the work conducted by examining development challenges that surface as a result of the interplay between different representations of software.

When examining the current challenges that exist in software development, maintenance, and testing one can observe a common trend throughout, contributing to a myriad of interconnected difficulties related to instantiating mental models of software into various functional representations. Incidentally, this thread stems from the hierarchy of abstraction discussed at the outset of this paper. In their text "*Foundations of Computer Science*" Aho and Ullman state that "*fundamentally, computer science is a science of abstraction – creating the right model for thinking about a problem and devising the appropriate mechanizable techniques to solve it.*" One could argue that software engineering is fundamentally centered upon effectively navigating various levels of an abstraction hierarchy, more specifically, the jump in abstraction between mental models and code. There is an inherent cost in reasoning between different levels of abstraction, generally measured in time and mental effort. In turn, the mental labor of such reasoning underlies many of the unique challenges experienced by today's developers.

However, it is not only effort in reasoning between *levels* of abstraction that can be difficult, but also between representations of the same abstraction level. Namely, foundational abstractions between *languages* that instantiate mental models of software can prove to be particularly troublesome. Here when we refer to the notion of a language we are not targeting programming languages specifically, but rather the broader definition of language as *a medium by which an idea or information is conveyed* [222]. In this sense, there are several different languages, or modalities, of information that developers must navigate during the software development process, including natural language and code, just to name a few. In essence, the bridging of the knowledge gap between these informa-

tion modalities constitutes a set of principal challenges related to *program comprehension* in software development.

Specific challenges in software engineering often stem from difficulties navigating different pairs of languages. For instance, when considering challenges related to software traceability, developers must reason between program representations related to natural language and code, interpreting how concepts and functional specifications dictated in natural language are dispersed throughout a codebase. When designing the graphical user interface of program, designers and developers must reason between the modalities of code and pixel-based image representations of the app via the graphical user interface. These pairs of contrasting information modalities have been labeled as *language dichotomies* [222]. Developing solutions to help developers more effectively reason between various language dichotomies is a key factor in helping to overcome many program comprehensions challenges.

More specifically, a language dichotomy can be defined as *a difficulty in program comprehension resulting from reasoning about different representations or modalities of information that describe a program* [222]. There are several language dichotomies that contribute to a varied set of problems. In the presented dissertation we focus on three different modalities of information:

1. ***Natural Language***: This modality represents languages that humans typically use to convey ideas or information to one another, such as English.
2. ***Code***: This modality represents the languages that humans utilize to construct a program, such as Java or Swift.
3. ***Visual Software Artifacts (GUIs)***: Much of today's user facing software is graphical. This information modality is highly visual, consisting of pixel-based representations of a program typically comprised of a logical set of building blocks often referred to as GUI-widgets or GUI-components.

Each of the representations described above have their own powerful uses, often serving to represent a program according to certain goals. For example, a GUI is an extremely powerful representation of program code that allows a user to carry out computing tasks, abstracting away the complexity of the underlying algorithms. However, for a developer, it is often critical to effectively understand and navigate how information represented in one modality translates to another. This is, at its core, a *program comprehension* task. For instance, a developer must reason about how different parts of the GUI correspond to different sections of code in a mobile app. However, bridging this gap between representations can be an arduous task, and thus underlies many open problems in mobile software development.

We assert that these language dichotomies can be effectively bridged through *automation*, thus helping to overcome several resulting software development problems. More specifically, by building upon techniques related to program analysis, machine learning, and computer vision, techniques can be derived to help automatically translate information across modalities, or detect anomalies between corresponding program representations in a single modality.

1.3 Research Context: Mobile Applications

In order to devise new approaches that automate the various components of the software development lifecycle, we need a suitable domain within which we can instantiate and evaluate them. In the scope of this dissertation, we focus our efforts on *mobile applications*. Mobile applications, often referred to colloquially as “apps”, are quite simply software applications that run on mobile hardware such as smartphones or tablets. Choosing mobile applications as our research domain is beneficial for at least the following three reasons: (i) there are open challenges unique to the software development process for mobile apps, providing a fertile research landscape, (ii) mobile apps, and by extension mobile app development, are *extremely* popular, giving our work a large potential for practical impact,

and (iii) mobile platforms provide a wide array of frameworks and utilities that facilitate varying types of program analysis. Background regarding mobile apps and mobile app development are detailed in Chapter 2. It should be noted that the work carried out in this dissertation is instantiated for the Android platform, mainly due to its open source nature and the litany of supporting tools and frameworks surrounding the platform. However, there are no substantial technical barriers that prevent the techniques presented in this dissertation from being transferred to other platforms.

1.4 Contributions & Outline

The core thesis of this dissertation is as follows:

Automating the process of instantiating and reasoning about the representation of conceptual software constructs in code, natural language, and visualizations allows for more effective software development by enabling rapid prototyping, swift iteration, and organic growth as abstract concepts evolve.

To investigate this thesis, we develop and empirically evaluate models and approaches that aid in automating the design, implementation, and testing of mobile applications. In particular, these approaches focus on automating (i) the verification of visual GUI properties for mobile app user interfaces, (ii) the construction of the code for mobile app GUIs given a target mock-up, and (iii) GUI-based functional testing of mobile apps. The intellectual merit of work presented in this dissertation lies in two interconnected contributions. First, we derive and illustrate how to construct a set of novel models for representing various attributes of mobile apps using information extracted via program analysis techniques. Second, we demonstrate that these models and the underlying encoded information can be utilized, both directly and in combination with machine learning techniques, to make the development and testing process for mobile apps more effective and efficient. The work conducted in this dissertation was of a collaborative nature and a summary of individual research contributions for each of the presented projects can be found in Appendix [A](#).

In **Chapter 2** we provide background related to mobile software development practices, and mobile graphical user interfaces. We also discuss related work related to (i) detecting design violations in mobile apps, (ii) automatically prototyping mobile application GUIs, and (iii) automated GUI-based testing of mobile apps.

In **Chapter 3** we develop a new technique for detecting and reporting instances where the GUI of a mobile application does not adhere to its intended design specifications as stipulated in a mock-up. This technique receives as input two images with accompanying metadata, one for the mock-up and one screenshot of the implemented GUI, and generates a detailed report stipulating instances where the specifications of the mock-up were not properly implemented. Our approach generates a hierarchical models of the mock-up and implementation of a particular screen of an application’s GUI and relates this model to the pixel-based images using coordinates. It then applies a computer vision technique called perceptual image differencing (PID) modeled after the human visual system to measure differences in images, and categorizes image differences according to an empirically derived taxonomy of GUI implementation errors. The content of this chapter is based primarily on the paper describing the GVT approach [224].

In **Chapter 4** we devise a technique to automate the process of translating an image-based mock-up of a mobile application’s GUI into suitable code. Our approach decomposes this translation process into three major steps: (i) detection of GUI elements, (ii) classification of these GUI elements into domain-specific, programmatic categories, and (iii) the construction and assembly of these categorized GUI elements into hierarchical code representation. Techniques from computer vision are utilized to *detect* GUI elements in image-based representations of GUIs. A deep convolutional neural network (CNN) trained on an automatically derived ground truth from tens of thousands applications screens is utilized to generate a model that is capable of accurately *classify* GUI elements into programmatic categories. Finally, we develop a data-driven k-nearest neighbors (KNN) algorithm for constructing realistic hierarchical representations of an app’s GUI before translating this representation into code. The content of this chapter is based primar-

ily on work describing the REDRAW approach [220] for automatically prototyping mobile application GUIs.

In **Chapter 5** we present a novel approach for automated testing of mobile applications that implements part of the vision outlined in Chapter 4. Our approach develops a new technique for constructing an on-the-fly event flow model of an application using systematic GUI exploration. Furthermore, our approach is capable of analyzing an application both statically and dynamically, extracting program features more likely to induce crashes, and stress-testing these features according to one of several strategies. This work described in this chapter is based primarily on work on the CRASHSCOPE approach [219].

In **Chapter 6** we discuss three major avenues for future research motivated by the outcomes of the work described in this dissertation. Finally, we offer general conclusions that summarize the contributions of this dissertation.

In addition to the contributions outlined in this dissertation, the author has worked on a wide array of research topics in software engineering over the course of his career as a doctoral student including: (i) bug reporting [225, 226, 223], Test Case Prioritization [206, 207], Mutation Testing [201, 207, 228], and Mobile Security [130, 262].

Chapter 2

Background & Related Work

In this chapter we first provide background related to mobile software development and the makeup of mobile graphical user interfaces with a focus on the Android platform. We then survey relevant work related to each of the three major projects presented in this dissertation which were introduced in the previous chapter, namely GVT, REDRAW, and CRASHSCOPE.

2.1 A Brief Introduction to Mobile Software Development

Mobile computing has become a centerpiece of modern society. Smartphones and tablets continue to evolve at a rapid pace and the computational prowess of these devices is approaching parity with laptop and desktop systems for high-end mobile hardware. This facilitates new categories of engaging software that aim to improve the ease of use and utility of computing tasks. Currently, many modern mobile apps have practically the same features as their desktop counterparts and range in nature from games to medical apps. The global “app” economy is comprised of millions of apps and developers, and billions of devices and users. Additionally, commodity smartphones are ushering in a completely new population of users from developing markets, many of whom are using a computer and accessing the internet for the first time. These factors, combined with the ease of distributing mobile apps on marketplaces like Apple’s App Store [\[24\]](#) or Google

Play [50] have made the development of mobile software a major focus of engineers around the world. In fact, according to Stack Overflow’s 2018 survey of over 100,000 developers [1], nearly a quarter of respondents identified themselves as mobile developers.

Mobile applications are typically developed on top of an existing *mobile platform*. These platforms consist of several different parts and these parts can vary between platforms, however at a minimum usually include: (i) a kernel and an operating system (OS) that runs on mobile hardware such as a smartphone, (ii) an application framework consisting of a set of platform specific APIs and libraries, and (iii) a set of tools and software to aid in developing apps, including IDEs or user interface builders. Mobile apps are typically written using a target programming language supported for a particular platform (e.g., Java and Kotlin for Android, and Objective-C and Swift for iOS), in combination with the APIs from the platform’s application framework. There are a shrinking set of platforms upon which developers can create and publish their apps. These platforms include Android, iOS, BlackBerry 10¹, Firefox OS, Ubuntu Touch, and Windows 10 Mobile¹. However, currently Android and iOS comprise the majority of the market, accounting for 87.7% and 12.1% of the market share respectively for the 2nd quarter of 2017 [89].

2.1.0.1 Unique Aspects and Challenges of the Mobile Development Process

While the importance and prevalence of mobile in the modern software development ecosystem is clear, many of the unique attributes that make mobile platforms attractive to both developers and users contribute a varied set of challenges that serve as obstacles to producing high-quality software.

Platform Evolution and Instability: Generally, the software development lifecycle typically follows a cyclic set of activities that include (i) requirement engineering, (ii) design, (iii) development, (iv) testing, and (v) maintenance. Modern agile development practices typically iterate quickly through these activities with the goal of delivering working software in a continuous manner where features are added and bugs are fixed during each

¹Support will end at the end of 2019

iterative development cycle. However, the rapid evolution of mobile platforms shapes the mobile development process in unique ways. As mobile hardware evolves, platforms evolve to keep pace with technological advancements, and new more convenient software features and capabilities are included with each iteration. For instance, Android has had over 15 major version releases since its inception in 2008 that have dramatically reshaped the underlying platform APIs [215], leading to support for advanced features such as Augmented Reality (AR). This iterative process puts immense pressure on developers to evolve their apps with the mobile hardware and platforms to satisfy the expectations of users that their apps take advantage of the latest features [178, 170]. This pressure leads to accelerated development cycles with a focus on adapting to changes in platform APIs. Adapting to these changes can be difficult and may adversely affect app quality [200, 125]; because developers must cope with adding additional app functionality based on new platform features, or on fixing bugs that arise due to changes in APIs currently used in an app. This may detract from time that could be spent on other activities such as fixing general regressions, refactoring, or improving the performance of an app, while also leading to undue technical debt. Thus, platform evolution has a clear affect on mobile development.

GUI-Centric, Event Driven Applications: Perhaps one of the most important features of mobile devices is the ease of use provided by high-fidelity, touch-enabled displays. Users primarily interact with their smartphones, tablets, and wearable devices and by extension the apps that run on these devices, through a touchscreen interface. This means that mobile apps are centered around the graphical user interface, and are driven by touch events on this interface. While other types of apps such as web apps, are also heavily event-driven, the unique touch based gestures and interactivity provided by mobile apps help to shape the software design, development and testing processes in unique ways. For example, the user interface (UI) and user experience in mobile apps must be well-designed for an app to be successful in highly competitive marketplaces. As such design and development tools for constructing UIs are a core part of IDEs such as Xcode and Android Studio that to mobile developers.

The event-driven nature of mobile apps also impacts testing. While developers can test small pieces of their code using practices such as unit testing, ultimately, testing must be done through the GUI. Manually testing applications is a time consuming practice that is fundamentally at odds with the rapid pace of mobile development practices. Thus, mobile developers and testers will often utilize automation frameworks that either allow for reusable or fully automated test input generation.

Mobile App Marketplaces: The primary (and some cases only) method of distribution for mobile apps is through “app marketplaces” such as Google Play or Apple’s App Store. These digital storefronts are unique to mobile applications, in that they provide users with easy access to purchase, download, and update apps, while providing mechanisms for users to review apps and provide feedback to developers. In recent years, these marketplaces have become increasingly competitive as the number of available apps numbers in the millions. App marketplaces incentivize developers to ensure their apps are of the highest possible quality, and to take into account the feedback of users. Developers need to ensure the quality of their apps by adhering to proper platform design principles and performing extensive testing, or risk being passed over for competitors. Likewise developers need to react to feedback communicated through user reviews by gathering and updating requirements and subsequently improving their app’s implementation.

Market, Device, and Platform Fragmentation: The large and growing user base of smartphones and tablets is one of the most alluring aspects for many developers and companies hoping to reach users. Unfortunately, targeting these users can be difficult due to multiple levels of fragmentation. The first level of fragmentation is at the market-level, which is currently dominated by Android and iOS. Thus, developers hoping to reach the maximum number of users must target *both* of these platforms. Second, there is fragmentation at the device level [165], as there is a large and growing number of hardware options for consumers to choose from with more devices being introduced each year. Finally, there is platform fragmentation, as users on the same mobile platform may be running different versions of mobile OSes. For instance, the latest version of iOS, iOS11, is currently run-

ning on 65% of devices whereas iOS10 currently encompasses 28% of the install base [123]. However, in Android fragmentation is more severe, with the two latest versions of Google's OS, Android 8 and 7, make up only 1.1% and 28.5% of the Android install base respectively. In order to create effective apps, developers must ensure that their applications function properly across a wide combination of different platforms, devices, and platform versions. This can make the process of developing and testing mobile apps challenging, as developers need to maintain concurrent codebases and test across a dizzying array of target configurations.

Naturally, these difficulties have led to creation of platform-independent development tools such as Xamarin [113], where a single codebase can be compiled to multiple platforms, eliminating the need for parallel codebases. Alternatively, there exist tools and frameworks like Ionic [54] for creating *hybrid applications* which use a combination of web technologies that interface with underlying platform APIs. In addition to hybrid applications, another framework created by Facebook called React Native [79] facilitates the development of native mobile apps using javascript and React. Applications built using react native are fully native to the target platform, the framework simply assembles the native code according to the javascript written by a developer. All of these approaches can help ease the burden of fragmentation when creating mobile apps. However, multi-platform development solutions come with their own set of compromises. For instance, hybrid apps are known to suffer from performance issues in terms of user interface interactivity, which can frustrate users. Furthermore, frameworks like Xamarin or React Native require their own learning curve, and developers are highly dependent upon the multi-platform framework keeping up with the latest features of modern mobile platforms.

2.1.1 Android Development Tools and Frameworks

Due to the open-source nature of a majority of the code that underlies the Android platform, and Google's push towards making the platform an inviting one for developers, there

exists a wide range of development tools. These tools and frameworks underpin many of the approaches presented in this dissertation, thus, we briefly introduce them here.

2.1.1.1 Virtual Android Devices

Given that applications developed for a mobile device typically will not run directly on the desktop systems used to develop them, it is important to have a vehicle by which developers can quickly test and preview their applications. This is primarily carried out by virtual Android devices. Currently, the most popular solutions for running virtual Android devices are the following: (i) the standard Android emulator officially supported by Google [6], (ii) the Genymotion [45] emulator, and (iii) Virtual Machines based on the `androidx86` project [15]. These devices can be used to test apps against different device and platform configurations to help combat the daunting fragmented market. Depending upon the host hardware and the configuration of the virtual devices, several devices can be instantiated concurrently to speed up testing.

2.1.1.2 Android Debugging Bridge

In order to test and verify different application properties, developers need a way to interact with running devices in a programmatic manner. This is primarily accomplished through the Android Debugging Bridge, also referred to as `adb`. The `adb` serves as a connection to a device and facilitates a variety of different development actions such as installing or uninstalling apps. It also allows for opening a unix-like shell on the device where a plethora of device actions can be performed, such as capturing screenshots and manipulating files.

2.1.1.3 UI Automator Framework

Given that Android applications are largely driven by touch-based events performed on a GUI, developers need a tool that lets them extract and analyze information about a device's screen. The Android UI Automator framework, often referred to as `uiautomator` [14], facilitates this process by offering means to capture hierarchical representations of the

GUI-components (*i.e.*, widgets) displayed on a device’s screen by encoding this structure into `xml` files. These `xml` files can then be parsed and analyzed to infer properties of the GUI of an app running on a target device.

2.1.1.4 Android Logcat

The Android `logcat` [9] utility is a command-line tool that capture and is capable of dumping a system log of an Android device, including stack traces for application errors and system messages.

2.2 Fundamentals of Mobile Graphical User Interfaces

The first two projects presented in this dissertation are concerned with automating different aspects related to the construction of Graphical User Interfaces of mobile apps. Thus, to provide sufficient context to the reader, in this section we introduce the fundamental concepts that underpin mobile graphical user interfaces. These concepts include the logical building blocks that comprise modern mobile GUIs as well as the concepts of design violations (*DVs*), and presentation failures.

2.2.0.1 GUI-Components, GUI-Containers, and Screens

There are three main logical constructs that define the concept of the GUI of a mobile app: *GUI-components* (or *GUI-widgets*), *GUI-containers*, and *Screens*. A *GUI-component* is a discrete object with a set of attributes (such as size and location among others) organized according to *GUI-containers* associated with a particular *Screen* of an app. *GUI-Containers* are logical constructs that group *GUI-components* and define relative spatial properties. A *Screen* is an invisible canvas of a size corresponding to the physical screen dimensions of a mobile device. We define two types of screens, those created by designers using professional-grade tools like Sketch, and those collected from implemented apps at runtime. Each of these two types of Screens has an associated set of GUI-components

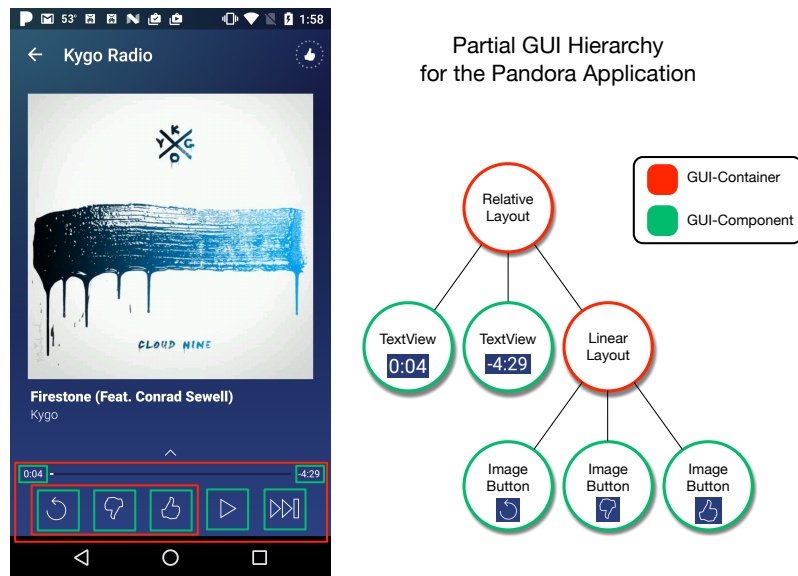


Figure 2.1: Illustration of the GUI Structure of the Pandora Android Application

(referred to interchangeably as *components* in this text). Each set of components associated with a screen is structured as a cumulative hierarchy comprising a tree structure, starting with a single root node, where the spatial layout of a parent always encompasses contained child components. These various GUI building blocks are illustrated in a partial hierarchy of the popular Pandora music application in Figure [2.1](#).

It is important to note that there are two parallel representations of mobile GUIs, *static* representations, and *dynamic* representations. There are key differences between these two representations that impact the definitions of key concepts described below. Static representations of a mobile GUI are those that are represented in code, usually in a domain specific language such as custom xml in Android, or nib files in iOS. Typically, the spatial attributes of a static representation of a mobile GUI are defined in *relative* terms, allowing for adaption to different screen sizes and device configurations. Dynamic representations of mobile GUIs are those that are rendered on a target device screen, and the spatial attributes of components are translated from their relative representation in the code to tangible coordinates that exist within the bounds of a target device's pixel-

based screen dimensions. The definitions presented below are amenable to either of these representations.

Definition 1 - GUI-Component: *Atomic graphical elements with pre-defined functionality, displayed within a GUI of a software application.*

More formally, a GUI-component is a discrete object GC with a corresponding set of attributes a which can be represented as a six-tuple in the form $(\langle x\text{-position}\rangle, \langle y\text{-position}\rangle, \langle height\rangle, \langle width\rangle, \langle text\rangle, \langle image\rangle)$. Here the first four elements of the tuple describe the location of the top left point for the bounding rectangle of the component, and the height and width attributes describe the size of the bounding box. These spatial elements can be represented either as relative values or as concrete pixel-based coordinates on a target device. The text attribute corresponds to text displayed by the component. Finally, the image attribute represents an image of the component with bounds adhering to the first four attributes. GUI-components have one of several domain dependent types, with each distinct type serving a different functional or aesthetic purpose. For example, common component types include dropdown menus and checkboxes, just to name a few. The notion of *atomicity* is important in this definition, as it differentiates GUI-components from *containers*. The third concept we define is that of a *GUI-container*:

Definition 2 - GUI-Container: *A logical construct that groups member GUI-components and typically defines spatial display properties of its members.*

In modern GUI-centric apps, GUI-components are rarely rendered on the screen using pre-defined coordinates. Instead, logical groupings of containers form hierarchical structures (or *GUI-hierarchies*). These hierarchies typically define spatial information about their constituent components, and in many cases react to changes in the size of the display area (*i.e., reactive design*)[\[12\]](#). For instance, a GUI-component that displays text may span the text according to the dimensions of its container.

Definition 3 - Screen: *A canvas with a predefined height and width corresponding to the physical display dimensions of a smartphone or tablet.*

Each Screen S contains a cumulative hierarchy of components, which can be represented as a nested set such that:

$$S = \{GC_1\{GC_2\{GC_i\}, GC_3\}\} \quad (2.1)$$

where each GC has a unique attribute tuple and the nested set can be ordered in either depth-first (Exp. 2.1) or in a breadth-first manner. Each nesting level in the set corresponds to a GUI-container, that logically groups these components. We are concerned with two specific types of screens: screens representing mock-ups of mobile apps S_m and screens representing real implementations of these apps, or S_r .

2.2.0.2 Design Violations & Presentation Failures

Now that we have introduced the fundamental building blocks of modern mobile GUIs, we next introduce the concepts related to design violations and presentation failures.

Definition 4 - Mock-Up Artifact: *An artifact of the software design and development process which stipulates design guidelines for GUIs and its content.*

In industrial mobile app development, mock-up artifacts typically come in the form of high fidelity images (with or without meta-data) created by designers using software such as Photoshop [4] or Sketch [86]. In this scenario, depending on design and development workflows, metadata containing information about the constituent parts of the mock-up images can be exported and parsed from these artifacts [2]. Independent developers may also use screenshots of existing apps to prototype their own apps. In this scenario, in addition to screenshots of running applications, runtime GUI-information (such as the html DOM-tree of a web app or the GUI-hierarchy of a mobile app) can be extracted to further aid in

²For example, by exporting Scalable Vector Graphics (.svg) or html formats from Photoshop.

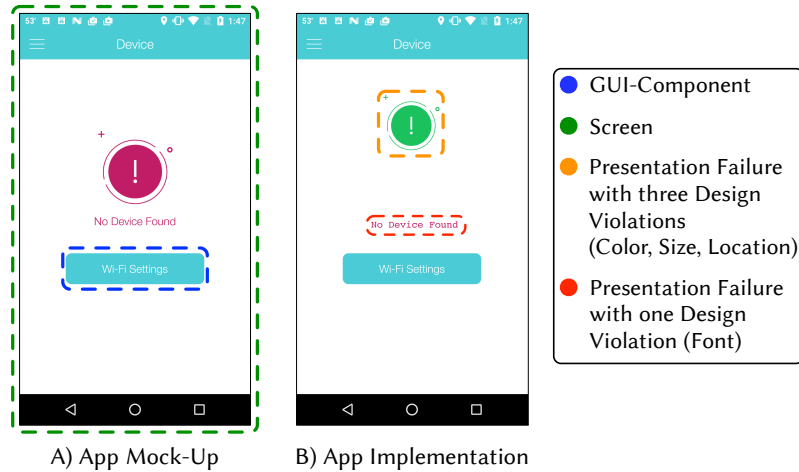


Figure 2.2: Examples of Formal Definitions

the prototyping process. However, this is typically *not* possible in the context of mock-up driven development, as executable apps do not exist.

Design violations correspond to visual symptoms of *presentation failures*, or differences between the intended design and implementation of a mobile app screen. Presentation failures can be made up of one or more design violations of different types.

Definition 5 - Design Violation: *A mismatch between at least one of the attributes of corresponding GUI-components existing in a mock-up artifact, and a dynamic representation of the implementation of that artifact.*

As shown in Exp. 2.2 a mismatch between the attribute tuples of two corresponding leaf-level (*e.g.*, having no direct children) GUI-components GC_{im} and GC_{jr} of two screens S_m and S_r imply a design violation DV associated with those components.

$$\begin{aligned}
 (GC_{im} \approx GC_{jr}) \wedge (GC_{im} \neq GC_{jr}) \\
 \implies DV \in \{GC_{im}, GC_{jr}\}
 \end{aligned}
 \tag{2.2}$$

In this definition leaf nodes correspond to one another if their location and size on a screen (*e.g.*, $\langle x\text{-position} \rangle$, $\langle y\text{-position} \rangle$, $\langle height \rangle$, $\langle width \rangle$) match within a given threshold. Equality between leaf nodes is measured as a tighter matching threshold across all attributes. Inequalities between different attributes in the associated tuples of the GCs lead to different types of design violations.

Definition 6 - Presentation Failure: *is a set of one or more design violations attributed to a set of corresponding GUI-components existing in a mock-up artifact, and a dynamic representation of the implementation of that artifact.*

Presentation Failures are a set of one or more design violations attributed to a set of corresponding GUI-components between two screens S_m and S_r , as shown in Exp. 3. For instance, a single set of corresponding components may have differences in both the $\langle x, y \rangle$ and $\langle height, width \rangle$ attributes leading to two constituent design violations that induce a single presentation failure PF . Thus each presentation failure between two Screens S corresponds to at least one mismatch between the attribute vectors of two corresponding leaf node GUI-components GC_{im} and GC_{ir} .

$$\begin{aligned} \text{if } \{DV_1, DV_2, \dots, DV_i\} \in \{GC_{im}, GC_{jr}\} \\ \text{then } PF \in \{S_m, S_r\} \end{aligned} \tag{2.3}$$

2.3 Work Related to Detection of GUI Design Violations in Mobile Apps

2.3.1 Detecting Presentation Failures in Web Applications

The most closely related work to GVT lies in approaches that aim at detecting, classifying and fixing presentation failures in web applications [209, 210, 242, 211]. Mahajan *et. al.* introduced WebSee aimed at this task. This approach leverages PID, clustering of difference regions, and localization of faulty html elements by resolving areas with visual discrepancies to content in an R-Tree representation of an html page. Mahajan *et. al.* also developed FieryEye, which builds upon the WebSee approach to identify faulty elements and corresponding styling properties. In comparison, GVT also performs detection and localization of presentation failures, but is the first to do so for mobile apps. In addition to the engineering challenges associated with building an approach to detect presentation failures in the mobile domain (*e.g.*, collection and processing of GUI-related data) GVT is

the first approach to leverage metadata from software mock-up artifacts (*e.g.*, Marketch) to perform *GC* matching based upon the spatial information collected from both mock-ups and dynamic application screens, allowing for precise detection of the different types of *DVs* delineated in our industrial *DV* taxonomy. *GVT* is also the first to apply the processes of *CQ*, *CH* analysis, and *B-PID* toward detecting differences in the content and color of icons and images displayed in mobile apps. *GVT* also identifies different faulty properties (such as location, color, or image content).

2.3.2 Cross-Browser Testing

Other approaches for *XBT* (or cross browser testing) by Roy Choudhry *et. al.* [242], [143], [244] examine and automatically report differences in web pages rendered in multiple browsers. These approaches are currently not directly applicable to mock-up driven development or mobile apps. An analogous problem in the domain of mobile apps is cross device testing for presentation failures (*e.g.*, ensuring proper rendering of GUIs across devices with different physical screen sizes and dimensions). However, this type of work is out of scope for the *GVT* approach, and left as a promising avenue for future work.

2.3.3 Other Approaches for GUI Verification

There are other approaches and techniques that are related to identifying problems or differences with GUIs of mobile apps. Xie *et al.* introduced *GUIDE* [267], a tool for GUI differencing that aims to describe discrepancies between successive releases of GUIs for an app by matching components between GUI-hierarchies. *GVT* utilizes a matching procedure for leaf node components using a similarity function based on spatial information, as direct tree comparisons are not possible in the context of mock-up driven development. Joorabchi *et. al* built an approach for detecting inconsistencies in multi-platform apps, but this was not related to mock-up driven development or GUIs specifically [179]. There has also been both commercial and academic work related to graphical software built specifically for creating high-fidelity mobile app mock-ups or mockups that encode information for auto-

mated creation of code for a target platform [217, 65, 44, 75]. However, such tools tend to either impose too many restrictions on designers who typically carry limited programming experience or do not allow for direct creation of code, thus *DVs* still persist in practice. Takashi [118] presented an approach for verifying GUI objects using the meta information stored in Windows graphics APIs and expected output from a set of program actions to verify GUI-objects. While this is a potentially useful approach, in the case of mobile mock-up driven development, sufficient GUI information can be gleaned via `uiautomator`.

2.4 Work Related to Automated Prototyping of Graphical User Interfaces for Mobile Apps

2.4.1 Reverse Engineering Mobile User Interfaces:

The most closely related research to the approach proposed in this paper is REMAUI, which aims to reverse engineer mobile app GUIs [232]. REMAUI uses a combination of Optical Character Recognition (OCR), CV, and mobile specific heuristics to detect components and generate a static app. The CV techniques utilized in REMAUI are powerful, and we build upon these innovations. However, REMAUI has key limitations compared to our work including: (i) it does not support the classification of detected components into their native component types and instead uses a binary classification of either text or images, limiting the real-world applicability of the approach, and (ii) it is unclear if the GUI-hierarchies generated by REMAUI are realistic or useful from a developer’s point of view, as the GUI-hierarchies of the approach were not evaluated.

In comparison, REDRAW (i) is not specific to any particular domain (although we implement our approach for the Android platform as well) as we take a data-driven approach for classifying and generating GUI-hierarchies, (ii) is capable of classifying GUI-components into their respective types using a CNN, and (iii) is able to produce realistic GUI-hierarchies using a data-driven, iterative KNN algorithm in combination with CV techniques. In our

evaluation, we offer a comparison of REDRAW to the REMAUI approach according to different quality attributes in Sections [4.3](#) & [4.4](#)

In addition to REMAUI, an open access paper (*i.e.*, non-peer-reviewed) was recently posted that implements an approach called pix2code [\[127\]](#), which shares common goals with REDRAW. Namely, the authors implement an encoder/decoder model that they trained on information from GUI-metadata and screenshots to translate target screenshots first into a domain specific language (DSL) and then into GUI code. However, this approach exhibits several shortcomings that call into question the real-world applicability of the approach: (i) the approach was only validated on a small set of synthetically generated applications, and no large-scale user interface mining was performed; (ii) the approach requires a DSL which will need to be maintained and updated over time, adding to the complexity and effort required to utilize the approach in practice. Thus, it is difficult to judge how well the approach would perform on real GUI data. In contrast, REDRAW is trained on a large scale dataset collected through a novel application of automated dynamic analysis for user interface mining. The data-collection and training process can be performed completely automatically and iteratively over time, helping to ease the burden of use for developers. To make for a complete comparison to current research-oriented approaches, we also include a comparison of the prototyping capability for real applications between REDRAW and the pix2code approach in Sections [4.3](#) & [4.4](#)

2.4.2 Mobile GUI Datasets

In order to train an accurate CNN classifier, REDRAW requires a large number of GUI-component images labeled with their domain specific types. To construct an effective classifier, we collect this dataset in a completely automated fashion by mining and automatically executing the top-250 Android apps in each category of Google Play excluding game categories, resulting in 14,382 unique screens and 191,300 labeled GUI-components (after data-cleaning). Recently, a large dataset of GUI-related information for Android apps, called RICO, was published and made available [\[151\]](#). This dataset is larger than

the one collected and described in this dissertation, containing over 72k unique screens and over 3M GUI-components. However, the REDRAW dataset is differentiated by some key factors specific to the problem domain of prototyping mobile GUIs:

1. ***Cropped Images of GUI-components:*** The REDRAW dataset of mobile GUI data contains a set of labeled GUI-components cropped from larger screenshots that are ready for processing by machine learning classifiers.
2. ***Cleaned Dataset:*** We implemented several filtering procedures at the app, screen, and GUI-component level to remove “noisy” components from the REDRAW dataset. This is an important factor for training an effective, accurate machine-learning classifier. These filtering techniques were manually verified for accuracy.
3. ***Data Augmentation:*** In the extraction of our dataset, we found that certain types of components were used more often than others, posing problems for deriving a *balanced* dataset of GUI-component types. To help mitigate this problem, we utilized data-augmentation techniques to help balance our observed classes.

We expand on the methodology for deriving the REDRAW dataset in Section [4.2.2.4](#). The RICO dataset does not exhibit the unique characteristics of the REDRAW dataset stipulated above that cater to creating an effective machine-learning classifier for classifying GUI-components. However, it should be noted that future work could adapt the data cleaning and augmentation methodologies stipulated in this project to the RICO dataset to produce a larger training set for GUI-components in the future.

2.4.3 Other GUI-Design and Reverse Engineering Tools:

Given the prevalence of GUI-centric software, there has been a large body of work dedicated to building advanced tools to aid in the construction of GUIs and related code [\[146, 134, 188, 139, 250, 216, 190\]](#) and to reverse engineer GUIs [\[137, 154, 153, 168, 253, 248\]](#).

While these approaches are aimed at various goals, they all attempt to reason logical, or programmatic info from graphical representations of GUIs.

However, the research projects referenced above exhibit one or more of the following attributes: (i) they do not specifically aim to support the task of automatically translating existing design mock-ups into code [154, 153, 168, 253, 248], (ii) they force designers or developers to compromise their workflow by imposing restrictions on how applications are designed or coded [250, 216, 188, 134, 146, 139] or (iii) they rely purely on reverse engineering existing apps using runtime information, which is not possible in the context of mock-up driven development [137, 216]. These attributes indicate that the above approaches are either not applicable in the problem domain which REDRAW aims to overcome (automatically generating application code from a mock-up artifact) or represent significant limitations that severely hinder practical applicability. Approaches that tie developers or designers into strict workflows (such as restricting the ways mock-ups are created or coded) struggle to gain adoption due to the competing flexibility of established image-editing software and coding platforms. Approaches requiring runtime information of a *target* app cannot be used in a typical mock-up driven development scenario, as implementations do not exist yet. While our approach relies on runtime data, it is collected and processed *independently* of the target app or mock-up artifact. Our approach aims to overcome the shortcomings of previous research by leveraging MSR and ML techniques to automatically infer models of GUIs for different domains, and has the potential to integrate into current design workflows as illustrated in Sec. 4.4.4

In addition to research on this topic, there are several commercial solutions which aim to improve the mock-up and prototyping process for different types of applications [65, 75, 44, 61, 73, 117, 64, 43, 56, 74, 55, 28]. These approaches allow for better collaboration among designers, and some more advanced offerings enable limited-functionality prototypes to be displayed on a target platform with support of a software framework. For instance, some tools will display screenshots of mock-ups on a mobile device through a preinstalled app, and allow designers to preview designs. However, these techniques *are not* capable of

translating mock-up artifacts into GUI code, and tie designers into a specific, potentially less flexible software or service. At the beginning of 2017, a startup has released software called Supernova Studio [92] that claims to be able to translate Sketch files into native code for iOS and Android. While this platform does contain some powerful features, such as converting Sketch screen designs into GUI code with “reactive” component coordinates, it exhibits two major drawbacks: (i) it is inherently tied to the Sketch application, and does not allow imports from other design tools, and (ii) it is not capable of classifying GUI-components into their respective types, instead relying on a user to complete this process manually [91]. Thus, REDRAW is complementary in the sense that our GUI-component classification technique could be used in conjunction with Supernova Studio to improve its overall effectiveness.

2.4.4 Image Classification using CNNs:

Large scale image recognition and classification has seen tremendous progress mainly due to advances in CNNs [186, 275, 255, 257, 167, 191]. These supervised ML approaches are capable of automatically learning robust, salient features of image categories from large numbers of labeled training images such as the ILSVRC dataset [246]. Building on top of LeCun’s pioneering work [191], the first approach to see a significant performance improvement over existing techniques (that utilized predefined feature extraction) was AlexNet [186], which achieved a top-5 mean average error (MAE) of $\approx 15\%$ on ILSVRC12. The architecture for this network was relatively shallow, but later work would show the benefits and tradeoffs of using deeper architectures. Zeiler and Fergus developed the ZFNet [275] architecture which was able to achieve a lower top-5 MAE than AlexNet ($\approx 11\%$) and devised a methodology for visualizing the hidden layers (or activation maps) of CNNs. More recent approaches such as GoogLeNet [257] and Microsoft’s ResNet [167] use deeper architectures (e.g., 22 and 152 layers respectively) and have managed to surpass human levels of accuracy on image classification tasks. However, the gains in network learning capacity afforded by deeper architectures come with a trade off in terms of training data

requirements and training time. In our experimental evaluation of REDRAW, we show that a relatively simple CNN architecture can be trained in a reasonable amount of time on popular classes of Android GUI-components, achieving a top-1 average classification accuracy of 91%.

2.5 Work Related to Automated Mobile Testing

In this subsection, we give an overview of the frameworks, tools, and services that are currently available to support mobile application testing, hinting at current limitations. In order to provide an “at-a-gance” overview of the current state of mobile testing, we summarize solutions currently available to developers (see Table 2.1). We focus on the use cases and existing problems and challenges with the state of the art.

We limit our analysis to research generally concerned with *functional* testing of mobile applications, and to popular commercial testing services and tools as gleaned from our previous research experience and industrial collaborations. The 7 categories of tools presented were derived in different ways. The first three categories (Automation Frameworks & APIs, Record & Replay Tools, and Automated Input Generation tools) have generally been defined by prior work [219, 142, 214], and we expand upon these past categorizations. The other four categories were derived by examining commercial software and service offerings available to mobile developers, as informed from our past experience. We delineated the features of these offerings, and it was clear that some tools shared common dimensions, thus forming the categories we present in this section.

2.5.1 Automation APIs/Frameworks

One of the most basic, yet most powerful testing tools available to developers on several mobile platforms are *GUI-Automation Frameworks* and *APIs* [14, 42, 25, 22, 82, 32, 38, 78, 84]. These tools often serve as interfaces for obtaining GUI-related information such as the hierarchy of components/widgets that exist on a screen and for simulating user interactions with a device. Because these frameworks and APIs provide a somewhat *universal interface*

Table 2.1: This Table Surveys the current state of tools, frameworks, and services that support activities related to mobile testing, originating from both Academic and Industrial backgrounds.

Automation Frameworks & APIs							
Name	GUI-Automation	OS API Automation	Black Box	Test-Case Recording	Cross-Device Support	Natural Language Test Cases	Open Source
UIAutomator [14]	Yes	No	Either	No	Limited	No	Yes
UIAutomation (iOS) [26]	Yes	No	No	Yes	Yes	No	Yes
Espresso [12]	Yes	No	No	No	Limited	No	Yes
Appium [22]	Yes	No	Yes	Yes	Limited	No	Yes
Robotium [84]	Yes	No	Yes	Yes	Limited	No	Yes
Robolectric [54]	No	Yes	No	No	Yes	No	Yes
Hamcrest [28]	No	No	Yes	Yes	Yes	No	No
Calabash [19]	Yes	No	No	No	No	Yes	Yes
Quantum [77]	Yes	N/A	No	N/A	N/A	Yes	No
Qmetry [76]	Yes	N/A	No	N/A	N/A	Yes	No
Record & Replay Tools							
	GUI Support	Sensor Support	Root Access Required	Cross-Device	High-Level Test Cases	Open Source	
RERAN [169]	Yes	No	Yes	No	No	No	
VALERA [171]	Yes	Yes	Yes	No	No	No	
Mosaic [164]	Yes	No	Yes	Limited	Yes	Yes	
Barista [165]	Yes	No	No	Yes	Yes	No	
Robotium Recorder [83]	No	No	No	Limited	Yes	No	
Xamarin Test Recorder [114]	Yes	No	No	Yes	Yes	No	
ODBR [223]	Yes	Yes	Yes	Limited	Yes	Yes	
SPAG-C [105]	Yes	No	N/A	N/A	No	No	
Espresso Recorder [11]	Yes	No	No	Limited	Yes	Yes	
Automated GUI-Input Generation Tools							
Tool Name	Instrumentation	GUI Exploration	Types of Events	Replayable Test Cases	NL Crash Reports	Emulators, Devices	Open Source
Random-Based Input Generation							
Monkey [43]	No	Random	System, GUI, Text	No	No	Both	Yes
Dynodroid [208]	Yes	Guided/Random	System, GUI, Text	No	No	Emulators	Yes
Intent Fuzzer [210]	No	Guided/Random	System (Intents)	No	No	N/A	No
VANARSena [230]	Yes	Random	System, GUI, Text	Yes	No	N/A	No
Systematic Input Generation							
AndroidRipper [121]	Yes	Systematic	GUI, Text	No	No	N/A	Yes
ACTEve [122]	Yes	Systematic	GUI	No	No	Both	No
A3E Depth-First [121]	Yes	Systematic	GUI	No	No	Both	Yes
CrashScope [219]	No	Systematic	GUI, Text, System	Yes	Yes	Both	No
Google RoboTest [48]	No	Systematic	GUI, Text	No	Yes	Devices	No
Model-Based Input Generation							
MobGUITar [120]	Yes	Model-Based	GUI, Text	Yes	No	N/A	Yes
A3E Targeted [121]	Yes	Model-Based	GUI	Yes	No	Both	No
SwiftHand [141]	Yes	Model-Based	GUI, Text	No	No	Both	Yes
QUANTUM [273]	Yes	Model-Based	System, GUI	Yes	No	N/A	No
ORBIT [268]	No	Model-Based	GUI	No	No	N/A	No
MonkeyLab [203]	No	Model-based	GUI, Text	Yes	No	Both	No
Zhang & Roumiev [278]	No	Model-based	GUI, Text	N/A	N/A	Both	Yes
Other Types of Input Generation Strategies							
PUMA [166]	Yes	Programmable	System, GUI, Text	No	No	Both	Yes
JPF-Android [260]	No	Scripting	GUI	Yes	No	N/A	Yes
CrashDroid [265]	No	Manual Rec/Replay	GUI, Text	Yes	Yes	Both	No
Collider [175]	Yes	Symbolic	GUI	Yes	No	N/A	No
SIG-Droid [218]	No	Symbolic	GUI, Text	Yes	No	N/A	No
Thor [110]	Yes	Test Cases	Test Case Events	N/A	No	Emulators	Yes
AppDoctor [170]	Yes	Multiple	System, GUI, Text	Yes	No	N/A	No
EvoDroid [212]	No	System/ Evo	GUI	No	No	N/A	No
Sapienz [214]	Yes	Search-Based	GUI, Text, System	Yes	Yes	Both	Yes
Jabbarvand et al. [176]	Yes	Search-Based	GUI, Text, System	Yes	Yes	Both	Yes
Bug & Error Reporting/Monitoring Tools							
	Video Recordings	App & GUI Analytics	Automatic Crash Reporting	Replayable Test Scripts	Open Source		
Airbrake [4]	No	No	Yes	No	No		
TestFairy [92]	Yes	No	Yes	No	No		
Appsee [26]	Yes	Yes	Yes	No	No		
BugCatcher [31]	Yes	No	No	No	No		
WatchSend [111]	Yes	Yes	Yes	No	No		
ODBR [223]	No	No	No	Yes	Yes		
FUSION [225, 228, 221]	No	No	No	Yes	Yes		
Testing Services							
	Crowdsourced Testing	Expert Testers	UX Testing	Functional Testing	Security Testing	Localization Testing	Open Source
Pay4Bugs [60]	Yes	N/A	No	Yes	No	N/A	No
TestArmy [92]	Yes	Yes	Yes	Yes	Yes	N/A	No
CrowdSourcedTesting [39]	Yes	Yes	Yes	Yes	Yes	No	No
CrowdSprint [97]	Yes	Yes	Yes	Yes	Yes	No	No
MyCrowdQA [92]	Yes	Yes	Yes	Yes	No	Yes	No
99Tests [3]	Yes	Yes	Yes	Yes	Yes	Yes	No
Applause [23]	Yes	Yes	Yes	Yes	Yes	Yes	No
Test.io [112]	Yes	Yes	Yes	Yes	N/A	N/A	No
Userlytics [104]	Yes	Yes	Yes	No	No	No	No
TestFlight [100]	Yes	No	No	Yes	No	N/A	No
SWRVE [33]	Yes	N/A	Yes (A/B Testing)	No	No	No	No
Loop11 [50]	Yes	No	Yes	No	No	No	No
Azetone [50]	Yes	No	Yes	No	No	No	No
UserZoom [105]	Yes	Yes	Yes	No	No	No	No
Apperian [20]	No	No	No	No	Yes	No	No
MrTappv [66]	N/A	N/A	Yes	N/A	N/A	N/A	N/A
LoadBack [45]	Yes	No	Yes	No	No	No	No
Appimize [22]	Yes	No	Yes (A/B Testing)	No	No	No	No
Cloud Testing Services							
	Automated Test Case Generation	Real Devices	Emulators	Remote Device Control	Test Reports	Open Source	
Xamarin Test Cloud [143]	No	Yes	No	No	Yes	No	
AWS Device Farm [29]	No	Yes	No	Yes	Yes	No	
Google Firebase [17]	Yes	Yes	No	No	Yes	No	
SauceLabs [83]	No	No	Yes	N/A	N/A	Partially	
TestGrid [101]	Yes	Yes	No	No	No	No	
Keynote [67]	No	Yes	No	Yes	Yes	No	
Perfecto [72]	No	Yes	No	Yes	Yes	No	
Birbar (TestDroid) [85]	Yes	Yes	No	No	No	No	
Device Streaming Tools							
	Streaming Over Internet	Streaming to Desktop from Connected Device	Recording	Open Source			
Vysor [102]	No	Yes	Yes (Screenshots)	No			
OpenSTF [30]	Yes	Yes	Yes (Screenshots)	Yes			
Appetize.io [21]	Yes	No	Yes (Video & Screenshots)	No			

to the GUI or underlying system functionality of a mobile platform, they typically underlie the functionality of many of the other input generation approaches and services discussed in this section. Typically these frameworks offer developers and testers an API for writing GUI-level tests for mobile apps through hand-written or recorded scripts. These scripts typically specify a series of actions that should be performed on different GUI-components (identifying them using varying attributes) and test for some state information via assertion statements. These frameworks are generally a good place to start for researchers who are interested in breaking into the mobile testing, as they are typically well documented and can offer a wealth of GUI and system related information which can be valuable for building tools or performing studies.

While useful for developers, these tools are not without their shortcomings. While these frameworks typically provide cross-device compatibility of scripts in *most* cases, there may be edge cases (*e.g.*, differing app states or GUI attributes) where scripts fail, highlighting the *fragmentation* problem. Also, they typically support only a *single testing objective*, as few tools offer support for complex user actions such as scrolling, pinching, or zooming or interfaces to simulate contextual states, which is required for effectively carrying out complex testing scenarios. More problematic, however, is that GUI level tests utilizing these frameworks are *very* expensive to maintain as an app evolves, discouraging many developers from adopting them in the first place. In the remainder of this subsection, we briefly outline the capabilities, pros, and cons of various Automation Frameworks and APIs for Android.

Both Google and Apple offer official “first-party” GUI-testing frameworks and APIs to allow developers to write tests to ensure their GUI is functioning as expected. Google’s open source `uiautomator` framework [14] allows for developers to write Junit-style GUI tests for Android multiple Android applications. The testing library makes use of the `uiautomator` framework included in the Android platform after version 4.3, and exposes APIs that developers and testers can use to write GUI-tests interacting with components identified by attributes such as the text they display. The `uiautomator` framework on an

Android device can also be accessed independently of GUI-tests. By interfacing with the framework through the `adb shell /system/bin/uiautomator dump` command, developers can access the view hierarchy of the currently displayed screen on a device or emulator. While `uiautomator` is a powerful tool, test scripts still suffer from issues such as expensive maintenance and limited cross-device compatibility. Google's *espresso* testing framework [42] allows for finer grained GUI-tests of a single target application, as opposed to the application independent tests provided by `uiautomator`. Additionally, the framework allows for the recording of test cases using the espresso test case recorder, and offers the ability for limited ui-control over web-views, leveraging the web-driver API. Espresso trades generality in the types of views that can be tested (*e.g.*, they must exist within your application) for finer grained control over the GUI views for tests, making it a better tool for testing a single app, whereas `uiautomator`'s flexibility allows for integration testing with multiple apps. iOS has a similar framework called *uiautomation* [25] that also allows for scripting and record-replay based GUI tests, that brings with it similar limitations. It should be noted that tests for these frameworks typically need to be compiled and bundled with an application to function properly, which can be prohibitive exercise for efficient development workflows.

In addition to the official frameworks described above there are also several third party solutions that for testing that offer deeper integrations with other services, or attempt to address shortcomings of the official solutions. Appium [22] is a cross-platform testing tool built on top of the Selenium WebDriver API that allows for the construction of UI-tests without modifying a subject application. This brings with it several advantages, including tests that can be written in a language of a developer or tester's choice, and a single ui-automation API that can be used across Android, iOS and hybrid applications. The Robotium [84] framework shares a similar design philosophy to Appium, but is Android specific. Robotium allows for black-box ui-tests that are more readable and robust due to run-time component binding [84], and it also integrates with popular Android build tools like Ant [17] and Gradle [51]. Roboelectric [82] is an Android specific tool that

allows for unit testing APIs included with the AndroidSDK without launching an app on an emulator. This is an attractive tool for developers as it allows for much faster testing, lending itself towards increasingly agile development practices. Calabash [32] is a testing framework for mobile apps maintained by Xamarin, and tightly coupled with Cucumber [38]. This framework allows for the creation of highly descriptive ui-acceptance tests without heavy modification of the underlying app. Xamarin also provides several cloud testing services around Calabash test cases. Ranorex [78] Is a UI test automation framework for mobile, web and desktop applications. It boasts several attractive features such as ease of use for non developers, advanced GUI object recognition and record replay functionality. Quantum [77] & QMetry [76] are testing frameworks that combines TestNG, Selenium WebDriver, Appium, and Perfecto (for cloud testing).

2.5.2 Record and Replay Tools

Manually writing test scripts for mobile GUI or system tests can be tedious, time consuming, and error prone. For this reason both academic and industrial solutions for *Record & Replay (R&R)* based testing have been devised. R&R is an attractive alternative to manually writing test scripts from an ease of use viewpoint, as it enables testers with very limited testing knowledge to create meaningful test scripts for apps. Additionally, some of the R&R approaches offer very fine grained (*e.g.*, millisecond accuracy) capture and replay of complex user actions, which can lend themselves well to testing scenarios which require such accuracy (*e.g.*, deterministically testing games) or portions of apps that require fine grained user input (*e.g.*, panning over a photo or a map).

However, despite the advantages and ease of use these types of tools afford, they exhibit several limitations. Most of these tools suffer from a trade-off between the timing and accuracy of the recorded events and the representative power and portability of recorded scripts. For context, some R&R-based approaches leverage the `/dev/input/event` stream situated in the linux kernel that underlies Android devices. While this allows for extremely accurate R&R, the scripts are usually coupled to screen dimensions and are agnostic to

the actual GUI-components with which the script interacts. This limits the possibility of *cross-device* R&R which would help alleviate the issue of testing across many different mobile devices. On the other hand, other R&R approaches may use higher-level representations of user actions, such as information regarding the GUI-components upon which a user acts. While this type of approach may offer more flexibility in easily recording test cases, it is limited in the accuracy and timing of events. An ideal R&R approach would offer the best of both extremes, both highly accurate and portable scripts, suitable for recording test cases or collecting crowdsourced data. R&R requires oracles that need to be defined by developers, by manually inserting assertions in the recorded scripts or using tool wizards. Thus, it is clear that this problematic dichotomy exacerbates more general mobile testing challenges including flaky tests, history agnostic test scripts, and fragmentation, and support for limited testing goals.

The first well-known mobile R&R approach, RERAN [159], came from academia. This approach records and translates actions captured from the linux kernel event stream at `/dev/input/event` and translates them events that can be re-injected into the event stream to replay the same series of actions later using Android's `getevent` and `sendevent` tools. This approach is extremely accurate and precise, both in terms of reproducing actions and the timing of those actions. However, it suffers from poor representativeness in that scripts are coupled to screen locations and are not suitable for cross-device or non-deterministic R&R scenarios. RERAN, was later extended in a tool called Mosaic [164] that claims to offer cross-device R&R for Android apps by mapping events to a normalized virtual screen and then scaling inputs linearly according to screen size. However, this approach will not work for all applications, as components do not always scale linearly with screen size, particularly for apps that run on both tablets and phones. The author's behind the RERAN approach also propose VALERA [171] tool which adopts a stream oriented approach for R&R and adds support for additional sensors such as the GPS. The approach essentially uses lightweight bytecode instrumentation to record events on the Android platform and replays them using an event log and injection into a particular Android app. Barista

[155], is a recently published tool that allows for device independent R&R of espresso test scripts, complete with assertion/oracle recording. However, currently this tool exhibits limitations in terms of the types of events it can record (*e.g.*, taps and long taps), and may fail to distinguish components under certain circumstances where components have similar attributes (*e.g.*, text, type). ODBR [223] is a recently proposed approach that run directly on an Android device and is capable of both fine grained user event recording and high-level script representation, making it highly suitable for bug recording and reproduction; however, it requires a rooted device or emulator to function properly. SPAG-C, is a R&R approach that uses the Sikuli [195] image recognition-based testing framework for R&R, and uses screenshots as image-based state-defining oracles. Each of these approaches has certain advantages and drawbacks and developer adoption of such tools is typically limited due to their inherent drawbacks.

In addition to these academic tools, there are several commercial solutions to mobile test-case R&R. The Robotium Recorder [83] allows for recording of test scripts in the Robotium testing language, with limited support for cross-device R&R. The Espresso Test Recorder [41] is a tool created by Google to allow for easy recording of espresso test scripts, complete with assertions, similar to Barista. It also shares the same limitations regarding the types of events that can be recorded. However, unlike Barista, it requires a device or emulator connected to a computer, rather than running as an app on the device itself. The Xamarin Test Recorder [114] is perhaps the most powerful and complete R&R tool available today, allowing for easy recording of test scripts and cross-device replay across devices in the Xamarin test cloud. However, this tool is currently in beta and is closed source, and ties developers into the Xamarin environment.

2.5.3 Automated Test Input Generation Techniques

Perhaps the most active area of mobile software testing research has been in the form of the *Automated Input Generation (AIG)* techniques. The premise behind such techniques is the following: Because manually writing or recording test scripts is a difficult, manual

practice, the process of input generation can be automated to dramatically ease the burden on developers and testers. Such approaches are typically designed with a particular goal, or set of goals in mind, such as achieving high code coverage, uncovering the largest number of bugs, reducing the length of testing scenarios or generating test scenarios that mimic typical use cases of an app. AIG approaches have generally been classified into three categories [142, 219]: *random-based* input generation [13, 208, 249, 239], *systematic* input generation [121, 122, 124, 219, 48], and *model-based* input generation [120, 124, 141, 273, 268, 203, 278]. Additionally, other input generation approaches have been explored including search-based and symbolic input generation [175, 218, 212, 214]. Nearly all of these approaches can trace their origins back to academic research, with companies like Google just recently entering the market with software-based automated testing services [48]. We provide at-a-glance information about these categories of approaches in Table 2.1

Research on this topic has made significant progress, particularly in the last few years, however, there are still persistent challenges. Recent work by Choudhary et. al. [142] illustrated the relative ineffectiveness of many research tools when comparing program coverage metrics against a naive random approach and highlighted many unsolved challenges including generation of system events, the cost of restarting an app, the need for manually specified inputs for certain complex app interactions, adverse side effects between different runs, a need for reproducible cases, mocking of services and inter-app communication, and a lack of support for cross-device testing scenario generation. While headway has been made regarding some of these challenges in recent work [219, 214], many have not been fully addressed. The specific limitations of these tools again fail to address broader challenges, including flaky tests, fragmentation, limited support for diverse testing goals, and inadequate developer feedback mechanisms.

2.5.3.1 Random/Fuzz Testing

Random input generation techniques, also commonly known as “fuzz testing” approaches rely on selecting arbitrary GUI or contextual events to generate input sequences that can be used for testing purposes. The most basic, and popular, form of this type of testing is using Google’s Android Monkey [13] tool. The tool allows for generation of a pseudo-random sequence of events to be generated for a connected device from the command line. Additionally developers can configure options such as the type of events to be generated, and the relative frequency of such events. However, while this is a good tool for fuzz/tress testing an application, the generated events are typically not replayable in a deterministic manner, the sequences generated are not indicative of how a human would use an app, and information about problems encountered are limited to stack traces. One of the first research-derived approaches for automated input generation, Dynodroid [208], maintains a history of event execution frequencies in a context-sensitive manner, and can more effectively generate new event sequences by biasing the generation algorithm toward or away from already executed events. Intent Fuzzer [249] is an approach that relies on static analysis to generate Android app intents, however, the approach has difficulty scaling with large apps due to the path-explosion problem. VanarSena [239] is a tool developed for Windows Phones that instruments application binaries to in order to test apps for faults caused by the injection of adverse contextual features. While these approaches can be effective in uncovering crashes, they are typically best suited for a single testing goal of “destructive-testing” [39] of applications, uncovering crashes along edges cases.

2.5.3.2 Ripping/Systematic Exploration

Another popular form of AIG approaches is that of *systematic* techniques. Tools employing this type of AIG strategy typically employ a hierarchal or tree representation of a GUI and exercise executable components that exist in this hierarchy according to a systematic traversal algorithm, such as depth or breadth-first search (BFS/DFS). While these testing

approaches can be effective at exploring significant portions of an application, they typically do not generate sequences typical of user-driven input and do not address many of the limitations pointed out at the beginning of this section.

Android Ripper [121] one of the first tools to employ this technique, dynamically derives a list of fireable event sequences for varying screens of an app, and then systematically executes the events in the generated lists if available. A³E [124] uses a combination of model-based and systematic-based input generation by employing static taint analysis to construct a high-level event-flow graph of an app complete with allowable transitions between screens. Using its systematic strategy, the tool performs a DFS over this event graph. ACTEve [122] is a concolic-based testing approach for Android that symbolically tracks events, but helps mitigate the path explosion problem by identifying subsuming event sequences. CrashScope [219] is a systematic-based AIG for Android that uses several combinations of execution strategies to elicit crashes from apps and generate expressive readable crash reports. Google’s Robo-Test [48] a recently released black-box tool that allows developers to upload their applications to be automatically tested by a systematic input generation approach. The exact exploration technique that Robo-Test uses is unknown, as the tool is not currently open source.

2.5.3.3 Model-Based Testing

Model-Based AIG approaches [120, 141, 273, 268, 203, 278] strive to derive a detailed, stateful model for an application under test and then generate input sequences allowing for most thorough coverage of that model. The models built by these approaches typically utilize static or dynamic analysis (or a combination of both) in order to properly construct the an application’s *state* and *event-flow*. The *state* of an application can consist of a variety of different parameters the GUI state and internal application state (*e.g.*, values of variables on the stack or heap). Most commonly in model-based AIG approaches, the state consists of several GUI-based attributes that identify unique screens. While test case generation based on these models can be a powerful tool, these models are rarely complete,

often only accounting for a small fraction of an app’s code coverage or feature set limiting their utility. Additionally, save for a few exceptions [203], model-based approaches tend not to closely emulate user behavior, and the models must be updated continually as the app changes, making maintenance expensive.

The *event-flow* of an app is typically defined as the allowable transitions between different states. including MobiGUITar [120] is an extension of Android Ripper that dynamically rips and models the the current state of an application through an *observe-model-exercise* paradigm. A³E targeted attempts to steer input generation toward targeted, unseen areas in an app’s statically derived event flow graph. Swifthand [141] uses an active learning approach to construct a model and generate inputs while striving to minimize app restarts by exploring all states accessible from the initial screen. QUANTUM [273] is a testing tool that generates Junit/Robotium test sequences for a specific app that include oracles. The authors of this paper defined several app agnostic oracles based on a study of common root causes of Android applications. ORBIT [268] uses static analysis to extract declared GUI components and link them to event-handlers to derive executable actions. It then constructs a hybrid model of an app using information from the previously performed static analysis and dynamic analysis to generate the model and input sequences. MonkeyLab [203] is an approach capable of mining application usages from users, modeling these application usages, and subsequently generating new events sequences based on the model. An approach by Zhang and Rountev [278] recently devised a model approach for testing notifications on Android Wear applications that typically run on smart watches or other wearable smart-devices.

2.5.3.4 Other Types of Input Generation Approaches

Additional types of AIG approaches are either geared toward input generation for a specific task (other than simply coverage or bug-finding capability) or utilize emerging underlying techniques for the task of input generation. These approaches include programmable automation frameworks like PUMA [166] and JPF android [260], approaches implementing

symbolic execution such as SIG-Droid [218] or Collider [175], and search-based approaches such as EvoDroid [212] or Sapienz [214]. Building advanced automation frameworks from the ground up is a difficult task, particularly in the context of research, and thus frameworks like PUMA and JPF-Android, typically lose out to the support of first-party automation libraries like uiautomator. Approaches employing symbolic and concolic execution are a promising development, however, they have typically only been demonstrated to enable robust test case generation on small applications, due to the curse of dimensionality in apps with many potential event sequences. Search-based techniques (both multi and single objective) have appeared as some of the most promising candidates for test generation thus far, however, still have challenges including generating tests for various tasks such as regression or use-case based testing.

PUMA [166] exposes high-level GUI-event to developers and testers, allowing for a programmable GUI-automation framework for which developers can implement their own exploration strategies. JPF-Android [260] is an extension of the Java PathFinder (JPF) tool to allow for running android-specific code directly on the JVM, similar to Roboelectric. It accomplishes this using stack manipulation, listeners and various logging techniques, but is limited in terms of the overhead these modifications cause and in the types of events it can properly execute. CrashDroid [265] is capable of translating a stack-trace from an Android application crash into expressive steps to reproduce a bug using a captured stack trace and manual annotated traces from users. Collider [175] is an approach for generating input event sequences that reach a targeted line of code in an Android application using a combination of concolic execution and a GUI model of an application. SIG-Droid [218] generates test inputs using symbolic execution combining inputs with a GUI-model extracted statically from the source code of an application. Thor [119] leverages existing test cases for an application and triggers adverse conditions for contextual features during the execution of these tests to simulate different environments in which an app may be used. AppDoctor [170] introduces an approach called "approximate execution" which relies on a side-loaded instrumentation app to execute event handlers associated with different

GUI-components, instead of triggering user-level actions on the components themselves. This approach speeds up execution time by sacrificing accuracy of the generated event sequences (as some sequences of triggered event handlers are impossible from the user-facing GUI of an app). EvoDroid and Sapienz [212, 214] introduce *search-based* approaches to automated input generation for Android applications. EvoDroid constructs a call-graph and GUI-model (denoted as “Interface” in the paper) and then uses an evolutionary genetic algorithm with a fitness function that attempts to maximize code coverage. Sapienz employs a multi-objective genetic algorithm capable of optimizing for code-coverage, sequence length, and number of crashes uncovered. Additionally, Sapienz uses an input generation approach based upon the idea of *motif patterns* that collect patterns of lower-level events that achieve higher coverage for longer event sequence generation. The approach devised by Jabbarvand *et al.* [173] allows for the dramatic minimization of existing test cases for Android applications while maintaining their ability to uncover energy bugs.

2.5.4 Bug and Error Reporting/Monitoring Tools

These types of tools have grown to become an integral part of many mobile testing workflows. There are two types of tools in this category: (i) tools for supporting bug reporting (a.k.a., issue trackers), and (ii) tools for monitoring crashes and resource consumption at run-time (*e.g.*, New relic [68] and Crashlytics [34]). Classic issue trackers only allow reporters to describe the bugs using textual reports and by posting additional files such as screenshots; but, real users can only report the bugs when an issue tracker is available for the app, as is the case of open source apps. In the case of tools for monitoring, if developers do not choose to include third-party error monitoring in their application (or employ a crowd-based approach), typically, the only user-feedback or in-field bug reports they receive are from user reviews or limited automated crash reports. Unfortunately, many user reviews or stack traces without context are unhelpful to developers, as they do not adequately describe issues with an application to the point where the problem can be reproduced and fixed. In order to mitigate these issues regarding visibility into applica-

tion bugs and errors, several tools and services exist that aim to help developers overcome this problem. These tools typically employ features that give developers more detailed information, such as videos [99, 26, 31, 111] or test scripts [223], on failures with concrete reproduction steps or stack traces (*e.g.*, crashes); however, to collect that information, the apps need to include API calls to the methods provided by the services. Additionally, they may provide analytic information about how users typically interact with an app, or assist end-users in constructing useful bug reports for apps [225, 226, 221]. Unfortunately, the automated error monitoring tools are limited to crash reporting (*i.e.*, exceptions and crashes), restricting their utility.

Airbrake [5] is a service for organizing and aggregating crash reports for an application. It helps developers and testers by grouping similar bugs into common groups and helps tracks the code quality of an app according the number and type of crashes reported against different versions. TestFairy [99], Appsee[26], BugClipper [31], and WatchSend [111] are services that allow a developer to include a third party library that reports logs, bug reports and video recordings which allow for better debugging of an app . In addition, Appsee offers developer UI-analytics (*e.g.*, touch heat-maps) that allows for insight into user behavior which can help improve the UI/UX experience as the app evolves. The On-Device Bug Reporting (ODBR) tool [223] allows for fine grained recording of end-user bug reports, complete with sensor streams. Unlike the services listed above, this app utilizes the fine-grained event collection of the `/dev/input/event` stream, similar to RERAN, but then translates these actions to higher-level representations of events, similar to MonkeyLab, with contextual information regarding UI-components with which the user interacted. This allows for *replayable* bug reports which can be captured by a user completely on a device. FUSION's [225, 226, 221] is an approach for off-device bug reporting that utilizes a combination of both static and dynamic analysis in order to help guide users through constructing useful bug reports by automatically suggesting steps for reproduction and including contextual information like screenshots.

2.5.5 Mobile Testing Services

Due to the sheer number of different technical challenges associated with automated input generation, and the typically high time-cost of manually writing or recording test scripts for mobile apps, *Mobile Testing Services* have become a popular alternative that utilize groups of human testing experts, or more general crowd-based workers. This allows the cost of test case generation or bug finding to be amortized across a larger group of workers compensated for their time devoted to testing. There are typically four different types of testing services offered including: (i) Traditional *Crowd-Sourced Functional Testing* [70, 97, 36, 37, 67, 3, 23, 102, 104, 100, 93, 59, 30, 105, 58, 27] which employs both experts and non-experts from around the world to submit bug reports relating to problems in apps, and who are compensated for the number of *true* bugs that are uncovered; (ii) *Usability testing* [36, 97, 37, 67, 105, 3, 102, 59, 30, 27] aims to measure the UX/UI design of an app with a focus on ease of use and intuitiveness; (iii) *Security Testing* [97, 23, 37, 3, 20], which aims to uncover any design flaws in an app that might compromise user security, and (iv) *Localization Testing* [36, 67, 3, 23], which aims to ensure that an app will function properly in different geographic regions with different languages across the world.

While these services do partially address some of the broader challenges of mobile testing such as fragmentation and support for limited testing goals, there are still several notable remaining challenges. None of these frameworks are open source or free, restricting developers from freely collecting critical usage data from the field which could improve general challenges such as test flakiness or history agnosticism by modeling collected information. Additionally, due to the time cost required of such crowdsourced services, they are typically not scalable in agile development scenarios where an app is constantly changing and released to customers.

In this section, rather than describing each service on its own, we summarize the benefits and challenges offered by each sub-type of service offered by popular companies.

2.5.5.1 CrowdSourced Functional Testing

Crowdsourced functional testing is the most popular type of service offered with most companies offering some form of the service [70, 97, 36, 37, 67, 3, 23, 102, 104, 100, 93, 59, 30, 105, 58, 27]. This type of service typically falls into one of two categories, expert-based or non-expert based. Expert-based testing services [97, 36, 37, 67, 3, 23, 102, 104, 105] typically employ developers or testers with general knowledge of the software testing and validation process, who are able to accurately report useful bug reports to developers before an app release. Other services recruit what development teams would generally classify as *end-users* [70, 100, 59, 30, 20, 58, 27] who may use the app in a fashion more representative of how a general population may use it, however because they are less experienced in the software testing process, may report many incomplete or unhelpful bug reports. Challenges presented by such services include dealing with large numbers of incomplete or duplicate bug reports submitted by testers, and limited communication or dialogue with testers after bug reports have been submitted.

2.5.5.2 Usability Testing

Testing the user experience and GUI-design is a particularly important facet of mobile app testing in general given the highly tactile and event driven nature of mobile platforms. Poor user experience can lead to decreased market share, or users abandoning an app in favor of similar alternatives. Mobile Usability Testing services are offered in a variety of forms. Some firms that offer these types of testing services will employ UX and design experts who offer feedback similar to a consultant [36, 97, 37, 67, 105]. Other services allow for one-on-one live conversations between developers and end-users or UX experts, that allow for feedback in the form of conversations or semi-structured interviews[58]. Other services and tools [3, 102, 59, 30, 105, 27] allow for recording of user sessions or UX specific crowd feedback of apps, including A/B testing, that allow development and design teams to understand user tendencies, and assess the design of their app accordingly.

2.5.5.3 Security Testing

In an age where privacy and security is at the forefront of many users minds given uncertain political climates and emerging legislation governing consumer electronics privacy, security testing and assurance will likely become a higher prioritized quality attribute for mobile developers. Several service offer solutions for security testing [97, 23], generally in the form of "security auditing" that examines different aspects of mobile app security including Improper platform usage, insecure communication between apps or the web, or potential authentication vulnerabilities. Additionally, services can employ "ethical hackers" that attempt to compromise an app and provide solutions to fix found vulnerabilities [37, 3]. Other services provide security features through APIs such as VPN or authentication services [20].

2.5.5.4 Localization Testing

Given the relatively low-cost of mobile smart devices, and the rapid rate at which they are gaining adoption across the world, it is unsurprising that when developing an app, it is important for developers to deploy their app to international marketplaces and ensure that it functions as expected across different locales. Testing of this type is typically referred to as "Localization Testing" and typically ensures app compatibility across different languages in terms of appearance and usability. Additionally, these services can provide insight into customs or tendencies specific to different locales so that developers can adjust features or functionalities of their app to cater accordingly. Several companies offer this service in the form of global groups of crowd-testers who are multilingual and accustomed to local tendencies of users [36, 67, 3, 23].

2.5.6 Cloud-Based Testing Frameworks

Due in part to the success of cloud providers who offer Software-as-a-Service (SaaS) products, several companies have launched *Cloud-Based Mobile Testing Services* [113, 101, 85,

[98] that allow developers to access hundreds of physical mobile devices arranged in “device farms” connected to cloud servers. Additionally, these services offer integration into development workflows such as running GUI-tests in a continuous integration fashion. This can help developers who have the means to pay for the often expensive services the ability to overcome the device fragmentation problem. Unfortunately, none of these services is open source and thus they are often out of reach for independent mobile developers or small teams.

Xamarin Test Cloud [113] integrates tightly into Xamarin’s suite of mobile development and testing tools and allows for the testing of applications on a large number of popular physical devices. It also supports playback of tests created in Xamarin’s integrated development environment. However, it does not allow for manual control of cloud-based physical devices. Amazon Web Services’ (AWS) device farm [29] is a similar offering, allowing developers direct access and manual control over cloud devices as well as automated fuzz testing with error reports. Google Firebase [47] offers remote execution of espresso and uiautomator test cases on physical devices attached to cloud servers. Testgrid [101] offers a large variety of end-to-end tools for effective mobile testing, ranging from a Test Case Writer, automated exploration, and a cloud device farm. Keynote [57] offers services that allow for both testing and monitoring of a mobile app, offering manual control of cloud-based physical devices. SauceLabs [85] and Perfecto Mobile [] offer similar testing and data aggregation services that allow for remote testing of mobile apps on real devices, and aggregated information relating to the executed tests. TestDroid [98] offers a variety of services ranging from remote test script execution to mobile-game testing based on Image Recognition techniques. While these services tend to exhibit attractive features, few offer easy to use, open source solutions accessible to independent developers.

2.5.7 Device Streaming Tools

Tools for *Device Streaming* can facilitate the mobile testing process by allowing a developer to mirror a connected device to their personal PC, or access devices remotely over the

internet. These tools can support use cases such as streaming secured devices to crowdsourced beta testers, or providing Q/A teams with access to a private set of physical or virtual devices hosted on company premises. They range in capabilities from allowing a connected device to be streamed to a local PC (Vysor [\[109\]](#)) to open source frameworks and paid services that can stream devices over the internet with low-latency (OpenSTF [\[90\]](#) & Appetize.io [\[21\]](#)) These tools, particularly OpenSTF, could support a wide range of important research topics that rely on collecting user data during controlled studies or investigations or tools related to crowdsourced testing.

Chapter 3

Automatically Reporting GUI Design Violations for Mobile Applications

Intuitive, elegant graphical user interfaces (GUIs) embodying effective user experience (UX) and user interface (UI) design principles are essential to the success of mobile apps. In fact, one may argue that these design principles are largely responsible for launching the modern mobile platforms that have become so popular today. Apple Inc's launch of the iPhone in 2007 revolutionized the mobile handset industry (heavily influencing derivative platforms including Android) and largely centered on an elegant, well-thought out UX experience, putting multitouch gestures and a natural GUI at the forefront of the platform experience. A decade later, the most successful mobile apps on today's highly competitive app stores (*e.g.*, Google Play [50] and Apple's App Store [24]) are those that embrace this focus on ease of use, and blend intuitive user experiences with beautiful interfaces. In fact, given the high number of apps in today's marketplaces that perform remarkably similar functions [63], the design and user experience of an app are often differentiating factors, leading to either success or failure [112].

Given the importance of a proper user interface and user experience for mobile apps, development usually begins with UI/UX design experts creating highly detailed mock-ups of app screens using one of several different prototyping techniques [254, 187]. The most

popular of these techniques and the focus of this project, is referred to as *mock-up driven development* where a designer (or group of designers) creates pixel perfect representations of app UIs using software such as Sketch [86] or PhotoShop [4]. Once the design artifacts (or *mock-ups*) are completed, they are handed off to development teams who are responsible for implementing the designs in code for a target platform. In order for the design envisioned by the UI/UX experts (who carry domain knowledge that front-end developers may lack) to be properly transferred to users, an accurate translation of the mock-up to code is *essential*.

Yet, implementing an intuitive and visually appealing UI in code is well-known to be a challenging undertaking [259, 231, 232]. As such, many mobile development platforms such as Apple’s Xcode IDE and Android Studio include powerful built-in GUI editors. Despite the ease of use such technologies are intended to facilitate, a controlled study has illustrated that such interface builders can be difficult to operate, with users prone to introducing bugs [274]. Because apps under development are prone to errors in their GUIs, this typically results in an iterative workflow where UI/UX teams will frequently *manually audit* app implementations during the development cycle and report any violations to the engineering team who then aims to fix them. This incredibly time consuming back-and-forth process is further complicated by several underlying challenges specific to mobile app development including: (i) continuous pressure for frequent releases [170, 178], (ii) the need to address user reviews quickly to improve app quality [237, 238, 144, 152], (iii) frequent platform updates and API instability [125, 200, 202, 215] including changes in UI/UX design paradigms inducing the need for GUI re-designs (*e.g.*, material design), and (iv) the need for custom components and layouts to support complex design mock-ups. Thus, there is a practical need for effective automated support to improve the process of detecting and reporting design violations and providing developers with more accurate and actionable information.

The difficulty that developers experience in creating effective GUIs stems from the need to manually bridge a staggering abstraction gap that involves reasoning concise and

accurate UI code from pixel-based graphical representations of GUIs. The GUI errors that are introduced when attempting to bridge this gap are known in literature as *presentation failures*. Presentation failures have been defined in the context of web applications in previous work as “a discrepancy between the actual appearance of a webpage [or mobile app screen] and its intended appearance” [210]. We take previous innovative work that aims to detect presentation errors in web applications [209, 210, 242, 143] as motivation to design equally effective approaches in the domain of mobile apps. Presentation failures are typically comprised of several *visual symptoms* or specific mismatches between visual facets of the intended GUI design and the implementation of those GUI-components [210] in an app. These visual symptoms can vary in type and frequency depending on the domain (*e.g.*, web vs. mobile), and in the context of mock-up driven development, we define them as *design violations*.

In this chapter, we present an approach, called GVT (**G**ui **V**erification **s**ys**T**em), developed in close collaboration with Huawei. Our approach is capable of automated, precise reporting of the design violations that induce presentation failures between an app mock-up and its implementation. Our technique decodes the hierarchal structure present in both mockups and dynamic representations of app GUIs, effectively matching the corresponding components. GVT then uses a combination of computer vision techniques to accurately detect design violations. Finally, GVT constructs a report containing screenshots, links to static code information (if code is provided), and precise descriptions of design violations. *GVT was developed to be practical and scalable, was built in close collaboration with the UI/UX teams at Huawei, and is currently in use by over one-thousand designers and engineers at the company.*

To evaluate the performance and usefulness of GVT we conducted three complementary studies. First, we empirically validated GVT’s *performance* by measuring the precision and recall of detecting synthetically injected design violations in popular open source apps. Second, we conducted a user study to measure the *usefulness* of our tool, comparing GVT’s ability to detect and report design violations to the ability of developers, while also

measuring the perceived utility of GVT reports. Finally, to measure the *applicability* of our approach in an industrial context, we present the results of an industrial case study including: (i) findings from a survey sent to industrial developers and designers who use GVT in their development workflow and (ii) semi-structured interviews with both design and development team managers about the impact of the tool. Our findings from this wide-ranging evaluation include the following key points: (i) In our study using synthetic violations GVT is able to detect design violations with an overall precision of 98% and recall of 96%; (ii) GVT is able to outperform developers with Android development experience in identifying design violations while taking less time; (iii) developers generally found GVT’s reports useful for quickly identifying different types of design violations; and (iv) GVT had a meaningful impact on the design and development of mobile apps for our industrial partner, contributing to increased UI/UX quality.

This chapter’s contributions can be summarized as follows:

- We formalize the concepts of *presentation failures* and *design violations* for mock-up driven development in the domain of mobile apps, and empirically derive common types of design violations in a study on an industrial dataset;
- We present a novel approach for detecting and reporting these violations embodied in a tool called GVT that uses hierarchal representations of an app’s GUI and computer vision techniques to detect and accurately report *design violations*;
- We conduct a wide-ranging evaluation of the GVT studying its *performance*, *usefulness*, and industrial *applicability*;
- We include an online appendix [\[52\]](#) with examples of reports generated by GVT and our evaluation dataset. Additionally, we make the GVT tool and code available upon request.

3.1 Problem Statement & Origin

In this section we formalize the problem of detecting *design violations* in GUIs of mobile apps and discuss the origin of the problem rooted in industrial mobile app design & development.

3.1.1 Problem Statement

At a high level, our goal is to develop an automated approach capable of detecting, classifying, and accurately describing *design violations* that exist for a single screen of a mobile app to help developers resolve *presentation failures* more effectively. In this section we formalize this scenario in order to allow for an accurate description and scope of our proposed approach. While this section focuses on concepts, Sec. [3.3](#) focuses on the implementation details.

3.1.1.1 Problem Statement

Taking into consideration the definitions introduced in Chapter [2.2](#), we can define the problem we aim to solve in this project more formally: Given two screens S_m and S_r corresponding to the mock-up and implementation screens of a mobile application, we aim to detect and describe the set of presentation failures $\{PF_1, PF_2, \dots, PF_i\} \in \{S_m, S_r\}$. We aim to report all design violations on corresponding GC pairs:

$$\begin{aligned} & \{DV_1, DV_2, \dots, DV_k\} \in \\ & \{\{GC_{i_1m}, GC_{j_1r}\}, \{GC_{i_2m}, GC_{j_2r}\}, \dots, \{GC_{i_xm}, GC_{j_yr}\}\} \end{aligned} \tag{3.1}$$

3.1.2 Industrial Problem Origins

A typical industrial mobile development process includes the following steps (as confirmed by our collaborators at Huawei): (i) First a team of designers creates highly detailed mockups of an app's screens using the Sketch [\[86\]](#) (or similar) prototyping software. These mock-ups are typically "pixel-perfect" representations of the app for a given screen dimen-

sion; (ii) The mock-ups are then handed off to developers in the form of exported images with designer added annotations stipulating spatial information and constraints. Developers use this information to implement representations of the GUIs for Android using a combination of Java and `xml`; (iii) Next, after the initial version of the app has been implemented, compiled Android Application Package(s) (*i.e.*, `apk`s) are sent back to the designers who then install these apps on target devices, generate screenshots for the screens in question, and manually search for discrepancies compared to the original mock-ups; (iv) Once the set of violations are identified, these are communicated back to the developers via textual descriptions and annotated screenshots at the cost of significant manual effort from the design teams. Developers must then identify and resolve the *DVs* using this information. The process is often repeated in several iterations causing substantial delays in the development process.

The goal of our work is to drastically improve this iterative process by: (i) automating the identification of *DVs* on the screens of mobile apps - saving both the design and development teams time and effort, and (ii) providing highly accurate information to the developers regarding these *DVs* in the form of detailed reports - in order to reduce their effort in resolving the problem.

3.2 Design Violations in Practice

In order to gain a better understanding of the types of *DVs* that occur in mobile apps in practice, we conducted a study using a dataset from Huawei. While there do exist a small collection of taxonomies related to visual GUI defects [192, 172] and faults in mobile apps [169, 201], we chose to conduct a contextualized study with our industrial partner for the following reasons: (i) existing taxonomies for visual GUI defects were not detailed enough, containing only general faults (*e.g.*, “incorrect appearance”), (ii) existing fault taxonomies for mobile apps either did not contain visual GUI faults or were not complete, and (iii) we wanted to derive a contextualized *DV* taxonomy for apps developed at Huawei. The

findings from this study underscore the existence and importance of the problem that our approach aims to solve in this context. Due to an NDA, we are not able to share the dataset or highlight specific examples, in order to avoid revealing information about future products at Huawei. However, we present aggregate results in this section.

3.2.1 Study Setting & Methodology

The *goal* of this study is to derive a taxonomy of the different types of *DVs* and examine the distribution of these types induced during the mobile app development process. The *context* of this study is comprised of a set of 71 representative mobile app mock-up and implementation screen pairs from more than 12 different internal apps, annotated by design teams from our industrial partner to highlight specific instances of *resolved DVs*. This set of screen pairs was specifically selected by the industrial design team to be representative both in terms of diversity and distribution of violations that typically occur during the development process.

In order to develop a taxonomy and distribution of the violations present in this dataset, we implement an open coding methodology consistent with constructivist grounded theory [138]. Following the advice of recent work within the SE community [256], we stipulate our specific implementation of this type of grounded theory while discussing our deviations from the methods in the literature. We derived our implementation from the material discussed in [138] involving the following steps: (i) establishing a research problem and questions, (ii) data-collection and initial coding, and (iii) focused coding. We excluded other steps described in [138], such as memoing because we were building a taxonomy of labels, and seeking new specific data due to our NDA limiting the data that could be shared. The study addressed the following research question: *What are the different types and distributions of GUI design violations that occur during industrial mobile app development processes?*

During the initial coding process, three of the authors were sent the full set of 71 screen pairs and were asked to code four pieces of information for each example: (i) a general category for the violation, (ii) a specific description of the violation, (iii) the severity of the

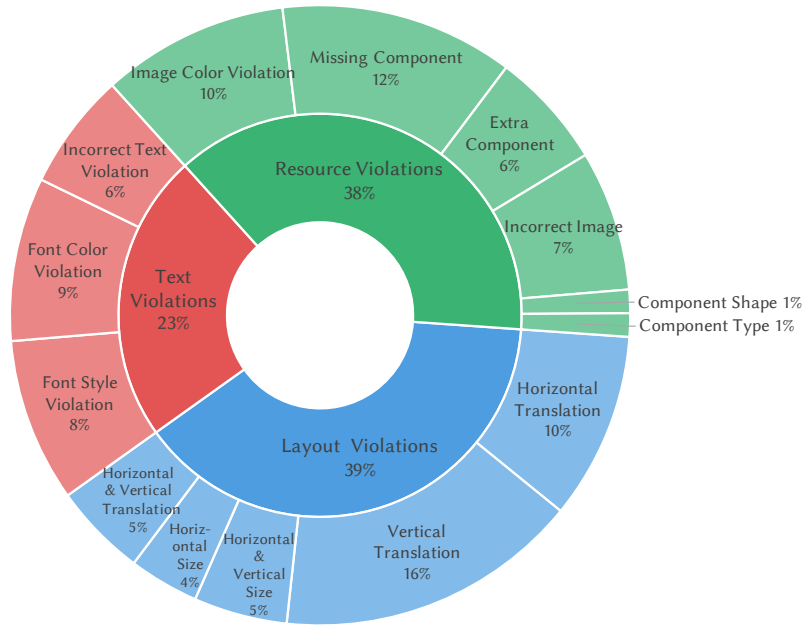


Figure 3.1: Distribution of Different Types of Industrial *DVs*

violation (if applicable), and (iv) the Android *GC* types affected (*e.g.*, button). Finally, we performed a second round of coding that combined the concepts of focused and axial coding as described in [138]. During this round two of the authors merged the responses from all three types of coding information where at least two of the three coders agreed. During this phase similar coding labels were merged (*e.g.*, “layout violation” vs. “spatial violation”), conflicts were resolved, two screen pairs were discarded due to ambiguity, and cohesive categories and subcategories were formed. The author agreement for each of the four types of tags is as follows: (i) general violation category (100%), (ii) specific violation description (96%), (iii) violation severity (100%), and (iv) affected *GC* types (84.5%).

3.2.2 Grounded Theory Study Results

Our study revealed three major categories of design violations, each with several specific subtypes. We forgo detailed descriptions and examples of violations due to space limitations, but provide examples in our online appendix [52]. The derived categories and subcategories of *DVs*, and their distributions, are illustrated in Fig. 3.1. Overall 82 *DVs* were identified across the 71 unique screen pairs considered in our study. The most preva-

lent category of *DVs* in our taxonomy are *Layout Violations* ($\approx 40\%$), which concern either a translation of a component in the x or y direction or a change in the component size, with translations being more common. The second most prevalent category ($\approx 36\%$) consists of *Resource Violations*, which concern missing components, extra components, color differences, and image differences. Finally, about one-quarter ($\approx 24\%$) of these violations are *Text Violations*, which concern differences in components that display text. We observed that violations typically only surfaced for “leaf-level” components in the GUI hierarchy. That is, violations typically only affected atomic components & not containers or backgrounds. Only 5/82 of examined violations ($\approx 6\%$) affected backgrounds or containers. Even in these few cases, the violations also affected “leaf-level” components.

The different types of violations correspond to different inequalities between the attribute tuples of corresponding GUI-components defined in Sec. [2.2](#). This taxonomy shows that designers are charged with identifying several different types of design violations, a daunting task, particularly for hundreds of screens across several apps.

3.3 The GVT Approach

3.3.1 Approach Overview

The workflow of GVT (Fig. [3.2](#)) proceeds in three stages: First in the *GUI-Collection Stage*, GUI-related information from both mock-ups and running apps is collected; Next, in the *GUI-Comprehension Stage* leaf-level *GCs* are parsed from the trees and a KNN-based algorithm is used to match corresponding *GCs* using spatial information; Finally, in the *Design Violation Detection Stage* *DVs* are detected using a combination of methods that leverage spatial *GC* information and computer vision techniques.

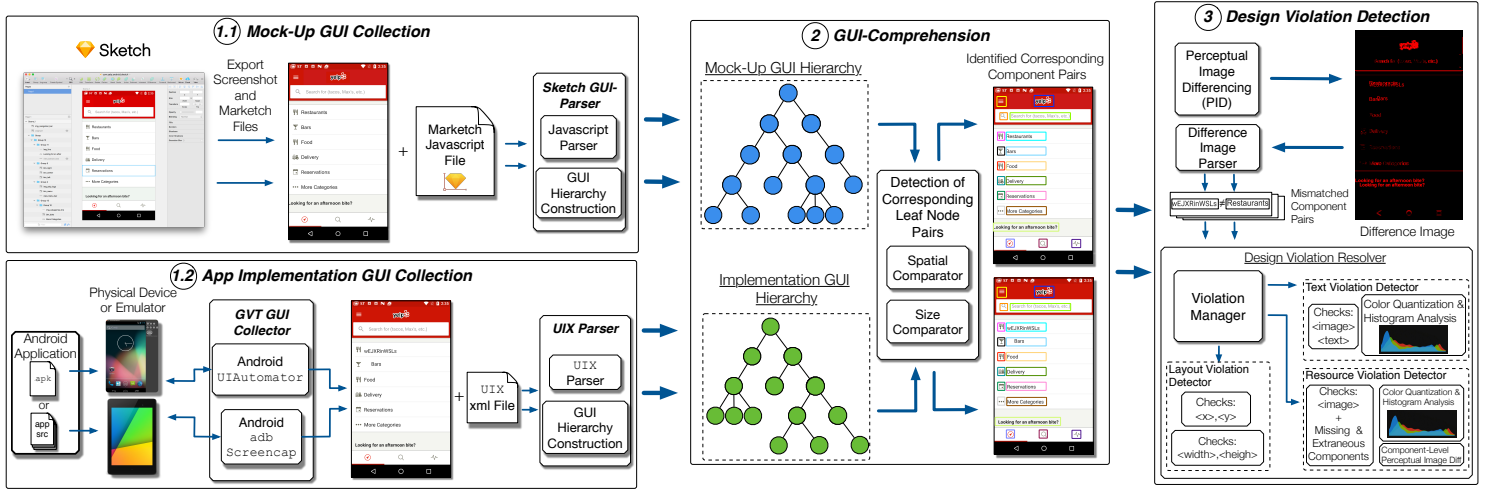


Figure 3.2: Overview of GVT Workflow

3.3.2 Stage 1: Mock-Up GUI Collection

3.3.2.1 Mock-Up GUI Collection

Software UI/UX design professionals typically use professional-grade image editing software (such as Photoshop [4] or Sketch [86]) to create their mock-ups. Designers employed by our industrial partner utilize the Sketch design software. Sketch is popular among mobile UI/UX designers due to its simple but powerful features, ease of use, and large library of extensions [87]. When using these tools designers often construct graphical representations of smartphone applications by placing *objects* representing *GCs* (which we refer to as *mock-up GCs*) on a *canvas* (representing a Screen S) that matches the typical display size of a target device. In order to capture information encoded in these mock-ups we decided to leverage an export format that was already in use by our industrial partner, an open-source Sketch extension called Marketch [60] that exports mock-ups as an `html` page including a screenshot and JavaScript file.

Thus, as input from the mock-up, GVT receives a screenshot (to be used later in the *Design Violation Detection Phase*) and a directory containing the Marketch information. The JavaScript file contains several pieces of information for each mock-up *GC* including, (i) the location of the mock-up *GC* on the canvas, (ii) size of the bounding box, and (iii)

the text/font displayed by the mock-up *GC* (if any). As shown in Figure 3.2(1.1), we built a parser to read this information. *However, it should be noted that our approach is not tightly coupled to Sketch or Marketch files*¹ After the Marketch files have been parsed, GVT examines the extracted spatial information to build a *GC* hierarchy. The result can be logically represented as a rooted tree where leaf nodes contain the atomic UI-elements with which a typical user might interact. Non-leaf node components typically represent containers, that form logical groupings of leaf node components and other containers. In certain cases, our approximation of using mock-up *GCs* to represent implementation *GCs* may not hold. For instance, an icon which should be represented as a single *GC* may consist of several mock-up *GCs* representing parts of the icon. GVT handles such cases in the *GUI-Comprehension* stage.

3.3.2.2 Dynamic App GUI-Collection

In order to compare the the mock-up of an app to its implementation GVT must extract GUI-related meta-data from a running Android app. GVT is able to use Android's `uiautomator` framework [14] intended for UI testing to capture `xml` files and screenshots for a target screen of an app running on a physical device or emulator. Each `uiautomator` file contains information related to the runtime GUI-hierarchy of the target app, including the following attributes utilized by GVT: (i) The Android component type (*e.g.*, `android.widget.ImageButton`), (ii) the location on the screen, (iii) the size of the bounding box, (iv) text displayed, (v) a developer assigned id. The hierarchal structure of components is encoded directly in the `uiautomator` file, and thus we built a parser to extract GUI-hierarchy using this information directly (see Fig. 3.2(1.2)).

¹Similar information regarding mock-up *GCs* can be parsed from the `html` or Scalable Vector Graphics (`.svg`) format exported by other tools such as Photoshop[4].

3.3.3 Stage 2: GUI Comprehension

In order for GVT to find visual discrepancies between components existing in the mock-up and implementation of an app, it must determine which components correspond to one another. Unfortunately, the GUI-hierarchies parsed from both the Marketch, and `uiautomator` files tend to differ dramatically due to several factors, making tree-based *GC* matching difficult. First, since the hierarchy constructed using the Marketch files is generated using information from the Sketch mock-up of app, it is using information derived from designers. While designers have tremendous expertise in constructing visual representations of apps, they typically do not take the time to construct programmatically-oriented groupings of components. Furthermore, designers are typically not aware of the correct Android component types that should be attributed to different objects in a mock-up. Second, the `uiautomator` representation of the GUI-hierarchy contains the runtime hierarchal structure of *GCs* and correct *GC* types. This tree is typically far more complex, containing several levels of containers grouping *GCs* together, which is required for the responsive layouts typical of mobile apps.

To overcome this challenge, GVT instead forms two collections of *leaf-node* components from both the mock-up and implementation GUI-hierarchies (Fig. 3.2-2), as this information can be easily extracted. As we reported in Sec. 3.2, the vast majority of *DVs* affects leaf-node components. Once the leaf node components have been extracted from each hierarchy, GVT employs a K-Nearest-Neighbors (KNN) algorithm utilizing a similarity function based on the location and size of the *GCs* in order to perform matching. In this setting, an input leaf-node component from the mock-up would be matched against its closest (*e.g.*, K=1) neighbor from the implementation based upon the following similarity function:

$$\gamma = (|x_m - x_r| + |y_m - y_r| + |w_m - w_r| + |h_m - h_r|) \quad (3.2)$$

Where γ is a similarity score where smaller values represent closer matches. The x, y, w and h variables correspond to the x & y location of the top and left-hand borders of

the bounding box, and the height and width of the bounding boxes for the mock-up and implementation *GCs* respectively. The result is a list of *GCs* that should logically correspond to one another (*corresponding GCs*).

It is possible that there exist instances of missing or extraneous components between the mock-up and implementation. To identify these cases, our KNN algorithm employs a *GC-Matching Threshold (MT)*. If the similarity score of the nearest neighbor match for a given input mock-up *GC* exceeds this threshold, it is not matched with any component, and will be reported as a *missing GC* violation. If there are unmatched *GCs* from the implementation, they are later reported as *extraneous GC* violations.

Also, there may be cases where a logical *GC* in the implementation is represented as small group of mock-up *GCs*. GVT is able to handle these cases using the similarity function outlined above. For each mock-up *GC*, GVT checks whether the neighboring *GCs* in the mockup are closer than the closest corresponding *GC* in the implementation. If this is the case, they are merged, with the process repeating until a logical GUI-component is represented.

3.3.4 Stage 3: Design Violation Detection

In the *Design Violation Detection* stage of the GVT workflow, the approach uses a combination of computer vision techniques and heuristic checking in order to effectively detect the different categories of *DVs* derived in our taxonomy presented in Section [3.2](#).

3.3.4.1 Perceptual Image Differencing

In order to determine corresponding *GCs* with visual discrepancies GVT uses a technique called Perceptual Image Differencing (PID) [\[270\]](#) that operates upon the mock-up and implementation screenshots. PID utilizes a model of the human visual system to compare two images and detect visual differences, and has been used to successfully identify visual discrepancies in web applications in previous work [\[209, 210\]](#). We use this algorithm in conjunction with the *GC* information derived in the previous steps of GVT to achieve ac-

curate violation detection. For a full description of the algorithm, we refer readers to [270]. The PID algorithm uses several adjustable parameters including: F which corresponds to the visual field of view in degrees, L which indicates the luminance or brightness of the image, and C which adjusts sensitivity to color differences. The values used in our implementation are stipulated in Section 3.3.5.

The output of the PID algorithm is a single *difference image* (Fig. 3.2-3) containing *difference pixels*, which are pixels considered to be perceptually different between the two images. After processing the difference image generated by PID, GVT extracts the implementation bounding box for each corresponding pair of GC s, and overlays the box on top of the generated difference image. It then calculates the number of difference pixels contained within the bounding box where higher numbers of difference pixels indicate potential visual discrepancies. Thus, GVT collects all “suspicious” GC pairs with a % of difference pixels higher than a *Difference Threshold DT*. This set of suspicious components is then passed to the *Violation Manager* (Fig. 3.2-3) so that specific instances of DVs can be detected.

3.3.4.2 Detecting Layout Violations

The first general category of DVs that GVT detects are *Layout Violations*. According to the taxonomy derived in Sec. 3.2 there are six specific layout DV categories that relate to two component properties: (i) screen location (*i.e.*, $\langle x,y \rangle$ position) and (ii) size (*i.e.*, $\langle h,w \rangle$ of the GC bounding box). GVT first checks for the three types of translation DVs utilizing a heuristic that measures the distance from the top and left-hand edges of matched components. If the difference between the components in either the x or y dimension is greater than a *Layout Threshold (LT)*, then these components are reported as a *Layout DV*. Using the LT avoids trivial location discrepancies within design tolerances being reported as violations, and can be set by a designer or developer using the tool. When detecting the three types of size DVs in the derived design violation taxonomy, GVT utilizes a heuristic that compares the width and height of the bounding boxes of corresponding

components. If the width or height of the bounding boxes differ by more than the LT , then a layout violation is reported.

3.3.4.3 Detecting Text Violations

The next general type of DV that GVT detects are *Text Violations*, of which there are three specific types: (i) Font Color, (ii) Font Style, and (iii) Incorrect Text Content. These detection strategies are only applied to pairs of text-based components as determined by `uiautomator` information. To detect font color violations, GVT extracts cropped images for each pair of suspicious text components by cropping the mock-up and implementation screenshots according to the component's respective bounding boxes. Next, *Color Quantization (CQ)* is applied to accumulate instances of all unique RGB values expressed in the component-specific images. This quantization information is then used to construct a *Color Histogram (CH)* (Fig. 3.2-3). GVT computes the normalized Euclidean distance between the extracted Color Histograms for the corresponding GC pairs, and if the Histograms do not match within a *Color Threshold (CT)* then a *Font-Color DV* is reported and the top-3 colors (i.e, centroids) from each CH are recorded in the GVT report. Likewise, if the colors do match, then the PID discrepancy identified earlier is due to the Font-Style changing (provided no existing layout DVs), and thus a Font-Style Violation is reported. Finally, to detect incorrect text content, GVT utilizes the textual information, preprocessed to remove whitespace and normalize letter cases, and performs a string comparison. If the strings do not match, then an *Incorrect Text Content DV* is reported.

3.3.4.4 Detecting Resource Violations

GVT is able to detect the following resource DVs : (i) missing component, (ii) extraneous component, (iii) image color, (iv) incorrect images, and (v) component shape. The detection and distinction between *Incorrect Image DVs* and *Image Color DVs* requires an analysis that combines two different computer vision techniques. To perform this analysis, cropped images from the mock-up and implementation screenshots according to corre-

sponding *GCs* respective bounding boxes are extracted. The goal of this analysis is to determine when the content of image-based *GCs* differ, as opposed to only the colors of the *GCs* differing. To accomplish this, GVT leverages PID applied to extracted *GC* images converted to a binary color space (*B-PID*) in order to detect differences in *content* and CQ and CH analysis to determine differences in *color* (Sec. [3.3.4.3](#)). To perform the B-PID procedure, cropped *GC* images are converted to a binary color space by extracting pixel intensities, and then applying a binary transformation to the intensity values (*e.g.*, converting the images to intensity independent black & white). Then PID is run on the color-neutral version of these images. If the images differ by more than an *Image Difference Threshold (IDT)*, then an *Incorrect Image DV* (which encompasses the *Component Shape DV*) is reported. If the component passes the binary PID check, then GVT utilizes the same CQ and CH processing technique described above to detect *image color DVs*. Missing and extraneous components are detected as described in Sec. [3.3.3](#)

3.3.4.5 Generating Violation Reports

In order to provide developers and designers with effective information regarding the detected *DVs*, GVT generates an `html` report that, for each detected violation contains the following: (i) a natural language description of the design violation(s), (ii) an annotated screenshot of the app implementation, with the affected GUI-component highlighted, (iii) cropped screenshots of the affected *GCs* from both the design and implementation screenshots, (iv) links to affected lines of application source code, (v) color information extracted from the CH for *GCs* identified to have color mismatches, and (vi) the difference image generated by PID. The source code links are generated by matching the `ids` extracted from the uiautomator information back to their declarations in the layout `xml` files in the source code (*e.g.*, those located in the `/res/` directory of an app's source code). We provide examples of generated reports in our online appendix [\[52\]](#).

3.3.5 Industrial Collaboration Methodology

Our implementation of GVT was developed in Java with a Swing GUI. In addition to running the GVT analysis the tool executable allows for one-click capture of `uiautomator` files and screenshots from a connected device or emulator. Several acceptance tests of mock-up/implementation screen pairs with pre-existing violations from apps under development within our industrial partner were used to guide the development of the tool. 12 Periodic releases of binaries for both Windows and Mac were made to deploy the tool to designers and developers within the company. The authors of this paper held regular bi-weekly meetings with members of the design and development teams to plan features and collect feedback.

Using the acceptance tests and feedback from our collaborators we tuned the various thresholds and parameters of the tool for best performance. The PID algorithm settings were tuned for sensitivity to capture subtle visual inconsistencies which are then later filtered through additional CV techniques: F was set to 45° , L was set to 100cdm^2 , and C was set to 1. The *GC-Matching Threshold (MC)* was set to $1/8\text{th}$ the screen width of a target device; the *DT* for determining suspicious *GCs* was set to 20%; The *LT* was set to 5 pixels (based on designer preference); the *CT* which determines the degree to which colors must match for color-based *DVs* was set to 85%; and finally, the *IDT* was set to 20%. GVT allows for a user to change these settings if desired, additionally users are capable of defining areas of dynamic content (*e.g.*, loaded from network activity), which should be ignored by the GVT analysis.

3.4 Design of the Experiments

To evaluate GVT's *performance*, *usefulness* and *applicability*, we perform three complimentary studies answering the following RQs:

- **RQ₁**: *How well does GVT perform in terms of detecting and classifying design violations?*
- **RQ₂**: *What utility can GVT provide from the viewpoint of Android developers?*
- **RQ₃**: *What is the industrial applicability of GVT in terms of improving the mobile application development workflow?*

RQ₁ and RQ₂ focus on quantitatively measuring the performance of GVT and the utility it provides to developers through a controlled empirical and a user study respectively. RQ₃ reports the results of a survey and semi-structured interviews with our collaborators aimed at investigating the industrial applicability of GVT .

3.4.1 Study 1: GVT Effectiveness & Performance

The *goal* of the first study is to quantitatively measure GVT in terms of its precision and recall in both detecting and classifying *DVs*.

3.4.1.1 Study Context

To carry out a controlled quantitative study, we manually reverse engineered Sketch mock-ups for ten screens for eight of the most popular apps on Google Play. To derive this set, we downloaded the top-10 apps from each category on the Google-Play store removing the various categories corresponding to games (as these have non-standard GUI-components that GVT does not support). We then randomly sampled one app from each of the remaining 33 categories, eliminating duplicates (since apps can belong to more than one category). We then manually collected screenshots and `uiautomator` files from two screens for each application using a Nexus 5, attempting to capture the “main” screen that a user would typically interact with, and one secondary screen. Using the `uiautomator` files, we generated cropped screenshots of all the leaf nodes components for each screen of the app. From these we were able generate 10 screens from 8 applications that successfully ran through GVT without any reported violations.

3.4.1.2 Synthetic DV Injection

With a set of correct mock-ups corresponding to implementation screens in an app, we needed a suitable method to introduce *DVs* into our subjects. To this end, we constructed a *synthetic DV injection tool* that modifies the `uiautomator xml` files and corresponding screenshots in order to introduce design violations from our taxonomy presented in Sec. 3.2. The tool is composed of two components: (i) an *XML Parser* that reads and extracts components from the screen, then (ii) a *Violation Generator* that randomly selects components and injects synthetic violations. We implemented injection for the following types of DVs:

Location Violation: The component is moved either horizontally, vertically, or in both directions within the same container. However, the maximum distance from the original point is limited by a quarter of the width of the screen size. This was based on the severity of Layout Violations in our study described in Section 3.2. In order to generate the image we cropped the component and moved it to the new location replacing all the original pixels by the most prominent color from the surroundings in the original location.

Size Violation: The component size either increases or decreases by 20% of the original size. For instances where the component size decreases, we replaced all the pixels by the most prominent color from the surroundings of the original size.

Missing Component Violation: This violation removes a leaf component from the screen, replacing the original pixels by the most prominent color from the surrounding background.

Image Violation: We perturb 40% of the pixels in an image by randomly generating an RGB value for the pixels affected.

Image Color Violation: This rule perturbs the color of an image by shifting the hue of image colors by 30°.

Component Color Violation: This uses the same process as for *Image Color Violations* but we change the color by 180°.

Font Violation: This violation randomly selects a font from the set of: *Arial*, *Comic Sans MS*, *Courier*, *Roboto*, or *Times Roman* and applies it to a `TextView` component.

Font Color Violation: changes the text color of a `TextView` component. We extracted the text color using CH analysis, then we changed the color using same strategy as for *Image Color Violations*.

3.4.1.3 Study Methodology

In injecting the synthetic faults, we took several measures to simulate the creation of realistic faults. First, we delineated 200 different types of design violations according to the distribution defined in our *DV* taxonomy in Sec. 3.2. We then created a pool of 100 screens by creating random copies of the both the `uiautomator xml` files and screenshots from our initial set of 10 screens. We then used the *synthetic DV injection tool* to seed faults into the pool of 100 screens according to the following criteria: (i) No screen can contain more than 3 injected *DVs*, (ii) each *GC* should have a maximum of 1 *DV* injected, and (iii) Each screen must have at least 1 injected *DV*. After the *DVs* were seeded, each of the 100 screens and 200 *DVs* were manually inspected for correctness. Due to the random nature of the tool, a small number of erroneous *DVs* were excluded and regenerated during this process (*e.g.*, color perturbed to perceptually similar color.). The breakdown of injected *DVs* is shown in Figure 3.3, and the full dataset with description is included in our online appendix 52.

Once the final set of screens with injected violations was derived, we ran GVT across these subjects and measured four metrics: (i) detection precision (*DP*), (ii) classification precision (*CP*), (iii) recall (*R*), and (iv) execution time per screen (*ET*). We make a distinction between detection and classification in our dataset because it is possible that GVT is capable of detecting, but misclassifying a particular *DV* (*e.g.*, an *image color DV* misclassified as an *incorrect image DV*). *DP*, *CP* and *R* were measured according to the following formulas:

$$DP, CP = \frac{T_p}{T_p + F_p} \quad R = \frac{T_p}{T_p + F_n} \quad (3.3)$$

where for DP , T_p represent injected design violations that were detected, and for CP , T_p represents injected violations that were both detected and classified correctly. In each case F_p correspond to detected DVs that were either not injected or misclassified. For Recall, T_p represents injected violations that were correctly detected and F_n represents injected violations that were not detected. To collect these measures, two authors manually examined the reports from GVT in order to collect the metrics.

3.4.2 Study 2: GVT Utility

Since the ultimate goal of an approach like GVT is to improve the workflow of developers, the *goal* of this second study is to measure the utility (*i.e.*, benefit) that GVT provides to developers by investigating two phenomena: (i) The accuracy and effort of developers in detecting and classifying DVs , and (ii) the perceived utility of GVT reports in helping to identify and resolve DVs .

3.4.2.1 Study Context

We randomly derived two sets of screens to investigate the two phenomena outlined above. First, we randomly sampled two mutually exclusive sets of 25, and 20 screens respectively from the 100 used in Study 1, ensuring at least one instance of each type of DV was included in the set. This resulted in both sets of screens containing 40 design violations in total. The correct mockup screenshot corresponding to each screen sampled from the study were also extracted, creating pairs of "correct" mockup and "incorrect" implementation screenshots. 10 participants with at least 5 years of Android development experience were contacted via email to participate in the survey.

3.4.2.2 Study Methodology

We created an online survey with four sections. In the first section, participants were given background information regarding the definition of DVs , and the different types of DVs derived in our taxonomy. In the second section, participants were asked about

demographic information such as programming experience and education level. In the third section, each participant was exposed to 5 mock-up/ implementation screen pairs (displayed side by side on the survey web page) and asked to identify any observed design violations. Descriptions of the *DVs* were given at the top of this page for reference. For each screen pair, participants were presented with a dropdown menu to select a type for an observed *DV*, and a text field to describe the error in more detail. For each participant, one of the 5 mock-up screens was a control, containing no injected violations. The 25 screens were assigned to participants such that each screen was observed by two participants and the order of the screens presented to each participant was randomized to avoid bias. To measure the effectiveness of participants in detecting and describing *DVs*, we leverage the *DP*, *CP* and *R* metrics introduced in Study 1. In the fourth section, participants were presented with two screen pairs from the second set of 20 sampled from the user study, as well as the GVT reports for these screens. Participants were then asked to answer 5 *user-preferences (UP)* and 5 *user experience (UX)* questions about these reports which are presented in the following section. The *UP* questions were developed according to the user experience honeycomb originally developed by Morville [229] and were posed to participants as free form text entry questions. We forgo a discussion of the free-form question responses due to space limitations, but we offer full anonymized participant responses in our online appendix [52]. We derived the Likert scale-based *UX* questions using the SUS usability scale by Brooke [132].

3.4.3 Study 3: Industrial Applicability of GVT

The *goal* of this final study is determine industrial applicability of GVT . To investigate this, we worked with Huawei to collect two sources of information: (i) the results of a survey sent to designers and developers who used GVT in their daily development/design workflow, and (ii) semi-structured interviews with both design and development managers whose teams have adopted the use of GVT.

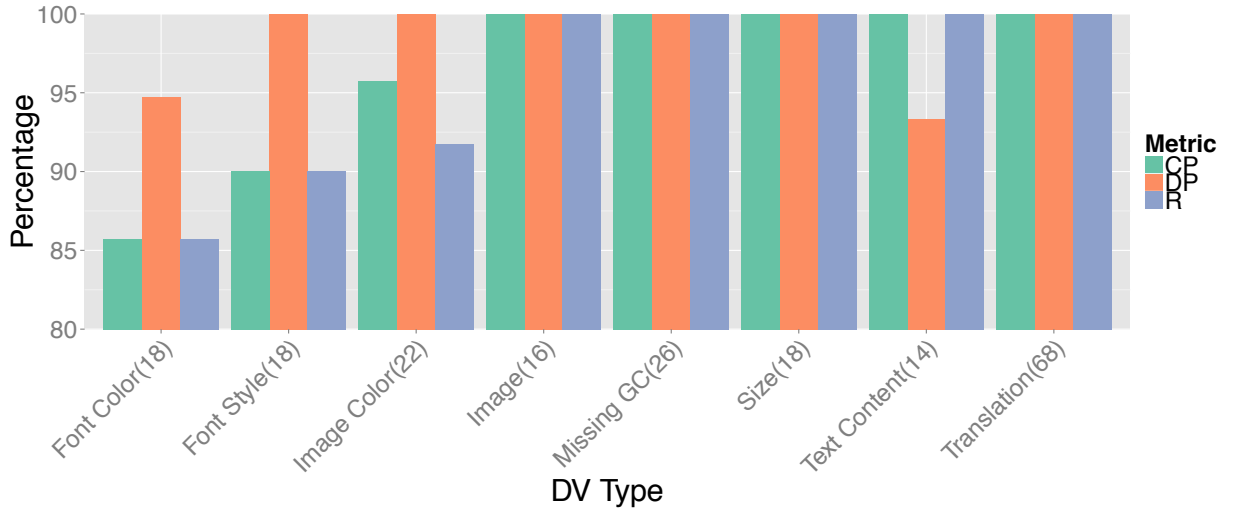


Figure 3.3: Study 1 - Detection Precision (DP), Classification Precision (CP), Recall(R)

3.4.3.1 Study Context & Methodology

We created a survey posing questions related to the *applicability* of GVT to industrial designers and developers. These questions are shown in Fig. 3.6. The semi-structured interviews were conducted in Chinese, recorded, and then later translated. During the interview, managers were asked to respond to four questions related to the *impact* and *performance* of the tool in practice. We include discussions of the responses in Section 3.5 and stipulate full questions in our appendix.

3.5 Empirical Results

3.5.1 Study 1 Results: GVT Performance

The results of Study 1, are shown in Figure 3.3. This figure shows the average DP , CP , and R for each type of seeded violation over the 200 seeded faults and the number of faults seeded into each category (following the distributions of our derived taxonomy) are shown on the x-axis. Overall, these results are extremely encouraging, with the overall DP achieving 99.4%, the average CP being 98.4%, and the average R reaching 96.5%. This illustrates that GVT is capable of detecting seeded faults designed to emulate both the type

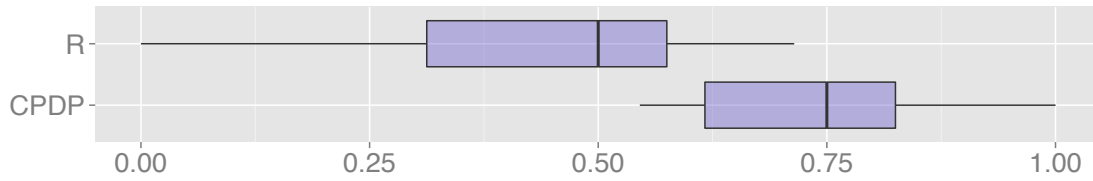


Figure 3.4: Study 2 - Developer CP, DP, and R

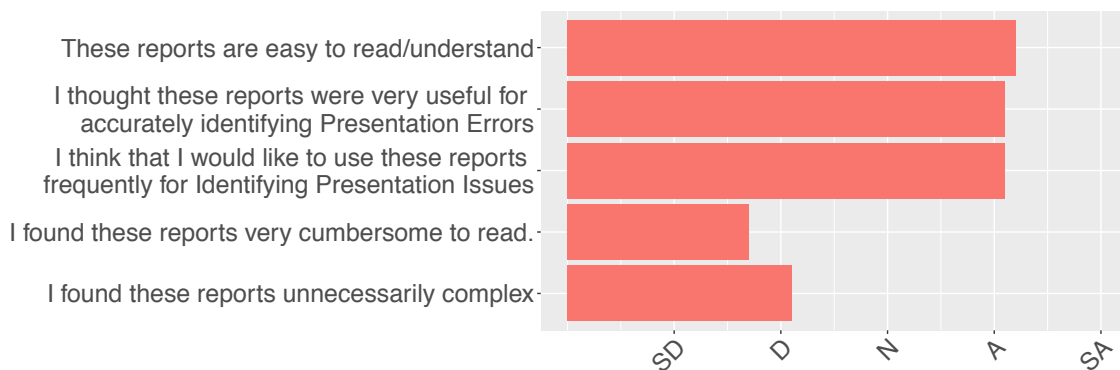


Figure 3.5: Study 2 - UX Question Responses. SD=Strongly Disagree, D=Disagree, N=Neutral, A=Agree, SA=Strongly Agree

and distribution of *DVs* encountered in industrial settings. While GVT achieved at least 85% precision for each type of seeded *DV*, it performed worse on some types of violations compared to others. For instance, GVT saw its lowest precision values for the *Font-Style* and *Font-Color* violations, typically due to the fact that the magnitude of perturbation for the color or font type was not large enough to surpass the Color or Image Difference Thresholds (*CT* & *IDT*). GVT took 36.8 mins to process and generate reports for the set of 100 screens with injected *DVs*, or 22 sec per screen pair. This execution cost was generally acceptable by our industrial collaborators.

3.5.2 Study 2 Results: GVT Utility

The *DP*, *CP* and *R* results, representing the Android developers ability to correctly detect and classify *DVs* is shown in Figure 3.4 as box-plots across all 10 participants. Here we found $CP=DP$, as when a user misclassified violations, they also did not detect them. As

this figure shows, the Android developers generally performed much worse compared to GVT achieving an average CP of under $\approx 60\%$ and an average R of $\approx 50\%$. The sources of this performance loss for the study participants compared to GVT was fourfold: (i) participants tended to report minor, acceptable differences in fonts across the examples (despite the instructions clearly stating *not* to report such violations); (ii) users tended to attribute more than one DV to a single component, specifically for *font style* and *font color* violations despite instructions to report only one; (iii) users tended to misclassify DVs based on the provided categories (*e.g.*, classifying a *layout DV* for a Text GC as an *incorrect text DV*), and (iv) participants missed reporting many of the injected DVs , leading to the low recall numbers. These results indicate that, at the very least, developers can struggle to both detect and classify DVs between mock-up and implementation screen pairs, signaling the need for an automated system to check for DVs before implemented apps are sent to a UI/UX team for auditing. This result confirms the notion that developers may not be as sensitive to small DVs in the GUI as the designers who created the GUI specifications. Furthermore, this finding is notable, because as part of the iterative process of resolving design violations, designers must communicate to developers DVs and developers must recognize and understand these DVs in order to properly resolve them. This process is often complicated due to ambiguous descriptions of DVs from designers to developers, or developers disagreeing with designers over the existence or type of a DV . In contrast to this fragmented process, GVT provides clear, unambiguous reports that facilitate communication between designers and developers.

Figure 3.5 illustrates the responses to the likert based UX questions, and the results are quite encouraging. In general, participants found that the reports from GVT were easy to read, useful for identifying DVs and indicated that they would like to use the reports for identifying DVs . Participants also indicated that the reports were not unnecessarily complex or difficult to read. We asked the participants about their preferences for the GVT reports as well, asking about the most and least useful information in the reports. *Every single* participant indicated that the highlighted annotations on the screenshots

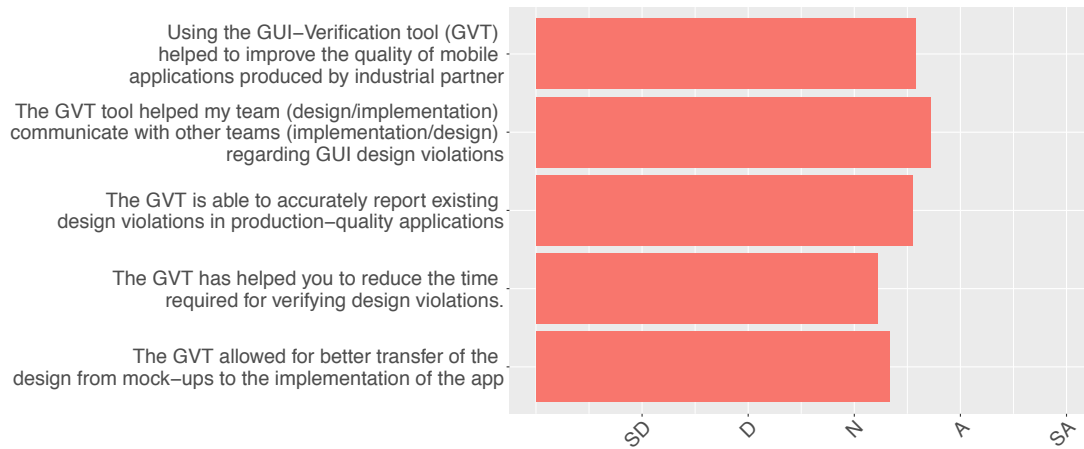


Figure 3.6: Study 3 - Applicability Questions. SD=Strongly Disagree, D=Disagree, N=Neutral, A=Agree, SA=Strongly Agree

in the report were the most useful element. Whereas most users tended to dislike the PID output included at the bottom of the report, citing this information as difficult to comprehend.

3.5.3 Study 3 Results: Industrial Applicability

The results for the *applicability* questions asked to 20 designers and developers who use GVT in their daily activities is shown in Figure 3.6. A positive outcome for each of these statements correlates to responses indicating that developers “agree” or “strongly agree”. The results of this study indicate a weak agreement of developers for these statements, indicating that while GVT is generally applicable, there are some drawbacks that prevented developers and designers from giving the tool unequivocal support. We explore these drawbacks by conducting semi-structured interviews.

In conducting the interviews, one of the authors asked the questions presented in Figure 3.6 to 3 managers (2 from UI/UX teams and 1 from a Front-End development team). When asked whether GVT contributed to an increased quality of mobile applications at the company, all three managers tended to agree that this was the case. For instance, one of the design managers stated, “*Certainly yes. The tool is the industry’s first*” and

the other designer manager added, *“When the page is more complicated, the tool is more helpful”*.

When asked about the overall performance and accuracy of the tool in detecting *DVs*, the manager from the implementation team admitted that the current detection performance of the tool is good, but suggested that dynamic detection of some components may improve it, stating, *“[DVs] can be detected pretty well... [but the tool is] not very flexible. For example, a switch component in the design is open, but the switch is off in the implementation”*. He suggested that properly handling cases such as this would make the tool more useful from a developers perspective. One of the design team managers held a similar view stating that, *“Currently, most errors are layout errors, so tool is accurate. Static components are basically detected, [but] maybe the next extension should focus on dynamic components.”* While the current version of the GVT allows for the exclusion of regions with dynamic components, it is clear that both design and development teams would appreciate proper detection of *DVs* for dynamic components. Additionally, two of the managers commented on the *“rigidity”* of the GVT’s current interface, and explained that a more streamlined UI would help improve its utility.

When asked about whether GVT improved communication between the design and development teams, the development team manager felt that while the tool has not improved communication yet, it did have the potential to do so, *“At present there is no [improvement] but certainly there is the potential possibility.”* The design managers generally stated that the tool has helped with communication, particularly in clarifying subtle *DVs* that may have caused arguments between teams in the past, *“If you consider the time savings on discussion and arguments between the two teams, this tool saves us a lot of time”*. Another designer indicated that the tool is helpful at describing *DVs* to developers who may not be able to recognize them with the naked eye *“We found that the tool can indeed detect something that the naked eye cannot”*. While there are certainly further refinements that can be made to GVT , it is clear that the tool has begun to have a positive impact of the

development of mobile apps, and as the tool evolves within the company, should allow for continued improvements in quality and time saved.

3.6 Limitations & Threats to Validity

Limitations: While we have illustrated that GVT is applicable in an industrial setting, the tool is not without its limitations. Currently, the tool imposes lightweight restrictions on designers creating Sketch mock-ups, chief among these being the requirement that bounding boxes of components do not overlap. Currently, GVT will try to resolve such cases during the *GUI-Comprehension stage* using an Intersection over union (IOU) metric.

Internal Validity: While deriving the taxonomy of *DVs*, mistakes in classification arising from subjectiveness may have introduced unexpected coding. To mitigate this threat we followed a set methodology, merged coding results, and performed conflict resolution.

Construct Validity: In our initial study (Sec. 3.2), a threat to construct validity arises in the form of the manner in which coders were exposed to presentation failures. To mitigate this threat, designers from our industrial partner manually annotated the screen pairs in order to clearly illustrate the affected *GCs* on the screen. In our evaluation of GVT threats arise from our method of *DV* injection using the *synthetic fault injection tool*. However, we designed this tool to inject faults based upon both the type and distribution of faults from our *DV* taxonomy to mitigate this threat.

External Validity: In our initial study related to the *DV* taxonomy, we utilized a dataset from a single (albeit large) company with examples across several different applications and screens. There is the potential that this may not generalize to other industrial mobile application development environments and platforms or mobile app development in general. However given the relatively consistent design paradigms of mobile apps, we expect the categories and the sub-categories within the taxonomy to hold, although it is possible that the distribution across these categories may vary across application development for differ-

ent domains. In Study 3 we surveyed employees at a single (though large) company, and findings may differ in similar studies at other companies.

3.7 Conclusion & Future Work

In the course of this project, we have formalized the problem of detecting design violations in mobile apps, and derived a taxonomy of design violations based on a robust industrial dataset. We presented GVT, an approach for automatically detecting, classifying, and reporting design violations in mobile apps, and conducted a wide ranging study that measured performance, utility, and industrial applicability of this tool. Our results indicate that GVT is effective in practice, offers utility for developers, and is applicable in industrial contexts.

Chapter 4

Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps

Most modern user-facing software applications are GUI-centric, and rely on attractive user interfaces (UI) and intuitive user experiences (UX) to attract customers, facilitate the effective completion of computing tasks, and engage users. Software with cumbersome or aesthetically displeasing UIs are far less likely to succeed, particularly as companies look to differentiate their applications from competitors with similar functionality. This phenomena can be readily observed in mobile application marketplaces such as the App Store [24], or Google Play [50], where many competing applications (also known as *apps*) offering similar functionality (*e.g.*, task managers, weather apps) largely distinguish themselves via UI/UX [112]. Thus, an important step in developing any GUI-based application is drafting and prototyping design mock-ups, which facilitates the instantiation and experimentation of UIs in order to evaluate or prove-out abstract design concepts. In industrial settings with larger teams, this process is typically carried out by dedicated designers who hold domain specific expertise in crafting attractive, intuitive GUIs using image-editing software such as Photoshop [4] or Sketch [86]. These teams are often responsible for expressing a

coherent design language across the many facets of a company’s digital presence, including websites, software applications and digital marketing materials. Some components of this design process also tend to carry over to smaller independent development teams who practice design or prototyping processes by creating wireframes or mock-ups to judge design ideas before committing to spending development resources implementing them. After these initial design drafts are created it is critical that they are faithfully translated into code in order for the end-user to experience the design and user interface in its intended form.

This process (which often involves multiple iterations) has been shown by past work and empirical studies to be challenging, time-consuming, and error prone [259, 231, 232, 193, 224] particularly if the design and implementation are carried out by different teams (which is often the case in industrial settings [224]). Additionally, UI/UX teams often practice an iterative design process, where feedback is collected regarding the effectiveness of GUIs at early stages. Using prototypes would be preferred, as more detailed feedback could be collected; however, with current practices and tools this is typically too costly [189, 230]. Furthermore, past work on detecting GUI design violations in mobile apps highlights the importance of this problem from an industrial viewpoint [224]. According to a study conducted with Huawei, a major telecommunications company, 71 unique application screens containing 82 design violations resulting from the company’s iterative design and development process were empirically categorized using a grounded-theory approach. This resulted in a taxonomy of mobile design violations spanning three major categories and 14 subcategories and illustrates the difficulties developers can have faithfully implementing GUIs for mobile apps as well as the burden that design violations introduced by developers can place on the overarching development process.

Many fast-moving startups and fledgling companies attempting to create software prototypes in order to demonstrate ideas and secure investor support would also greatly benefit from rapid application prototyping. Rather than spending scarce time and resources on iteratively designing and coding user interfaces, an accurate automated approach would

likely be preferred. This would allow smaller companies to put more focus on features and value and less on translating designs into workable application code. Given the frustrations that front-end developers and designers face with constructing accurate GUIs, there is a clear need for automated support.

To help mitigate the difficulty of this process, some modern IDEs, such as XCode [115], Visual Studio [108], and Android Studio [11], offer built-in GUI editors. However, recent research suggests that using these editors to create complex, high-fidelity GUIs is cumbersome and difficult [189], as users are prone to introducing bugs and presentation failures even for simple tasks [274]. Other commercial solutions include offerings for collaborative GUI-design and for interactive previewing of designs on target devices or browsers (displayed using a custom framework, with limited functionality) [65, 75, 44, 61, 73, 117, 64, 43, 56, 74, 55, 28], but none offer an end-to-end solution capable of automatically translating a mock-up into accurate native code for a target platform. It is clear that an automated tool capable of even partially automating this process could significantly reduce the burden on the design and development processes.

To help mitigate the difficulty of this process, some modern IDEs, such as XCode [115], Visual Studio [108], and Android Studio [11], offer built-in GUI editors. However, recent research suggests that using these editors to create complex, high-fidelity GUIs is cumbersome and difficult [189], as users are prone to introducing bugs and presentation failures even for simple tasks [274]. Other commercial solutions include offerings for collaborative GUI-design and for interactive previewing of designs on target devices or browsers (displayed using a custom framework, with limited functionality) [65, 75, 44, 61, 73, 117, 64, 43, 56, 74, 55, 28], but none offer an end-to-end solution capable of automatically translating a mock-up into accurate native code (with proper component types) for a target platform. It is clear that a tool capable of even partially automating this process could significantly reduce the burden on the design and development processes.

Unfortunately, automating the prototyping process for GUIs is a difficult task. At the core of this difficulty is the need to bridge a broad abstraction gap that necessitates reasoning accurate user interface code from either pixel-based, graphical representations of GUIs or digital design sketches. Typically, this abstraction gap is bridged by a developer’s domain knowledge. For example, a developer is capable of recognizing discrete objects in a mock-up that should be instantiated as components on the screen, categorizing them into proper categories based on their intended functionalities, and arranging them in a suitable hierarchical structure such that they display properly on a range of screen sizes. However, even for a skilled developer, this process can be time-consuming and prone to errors [224]. Thus, it follows that an approach which automates the GUI prototyping process must bridge this image-to-code abstraction gap. This, in turn, requires the creation of a model capable of representing the domain knowledge typically held by a developer, and applying this knowledge to create accurate prototypes.

Given that, within a single software domain, the design and functionality of GUIs can vary dramatically, it is unlikely that manually encoded information or heuristics would be capable of fully supporting such complex tasks. Furthermore, creating, updating, and maintaining such heuristics manually is a daunting task. Thus, we propose to learn this domain knowledge using a data-driven approach that leverages machine learning (ML) techniques and the GUI information already present in existing apps (specifically screenshots and GUI metadata) acquired via mining software repositories (MSR).

More specifically, we present an approach that deconstructs the prototyping process into the tasks of: *detection*, *classification*, and *assembly*. The first task involves *detecting* the bounding boxes of atomic elements (*e.g.*, GUI-components which cannot be further decomposed) of a user interface from a mock-up design artifact, such as pixel-based images. This challenge can be solved either by parsing information regarding objects representing GUI-components directly from mock-up artifacts (*e.g.*, parsing exported metadata from Photoshop), or using CV techniques to infer objects [232]. Once the GUI-components from a design artifact have been identified, they need to be *classified* into their proper

domain-specific types (*e.g.*, button, dropdown menu, progress bar). This is, in essence, an image classification task, and research on this topic has shown tremendous progress in recent years, mainly due to advancements in deep convolutional neural networks (CNNs) [186, 275, 255, 257, 167]. However, because CNNs are a supervised learning technique, they typically require a large amount of training data, such as the ILSVRC dataset [246], to be effective. We assert that automated dynamic analysis of applications mined from software repositories can be applied to collect screenshots and GUI metadata that can be used to *automatically* derive labeled training data. Using this data, a CNN can be effectively trained to classify images of GUI-Components from a mock-up (extracted using the detected bounding boxes) into their domain specific GUI-component types. However, classified images of components are not enough to *assemble* effective GUI code. GUIs are typically represented in code as hierarchal trees, where logical groups of components are bundled together in containers. We illustrate that an iterative K-nearest-neighbors (KNN) algorithm and CV techniques operating on mined GUI metadata and screenshots can construct realistic GUI-hierarchies that can be translated into code.

We have implemented the approach described above in a system called REDRAW for the Android platform. We mined 8,878 of the top-rated apps from Google Play and executed these apps using a fully automated input generation approach (*e.g.*, GUI-ripping) derived from our prior work on mobile testing [219, 203]. During the automated app exploration the GUI-hierarchies for the most popular screens from each app were extracted. We then trained a CNN on the most popular native Android GUI-component types as observed in the mined screens. REDRAW uses this classifier in combination with an iterative KNN algorithm and additional CV techniques to translate different types of mock-up artifacts into prototype Android apps. We performed a comprehensive set of three studies evaluating REDRAW aimed at measuring (i) the accuracy of the CNN-based classifier (measured against a baseline feature descriptor and Support Vector Machine based technique), (ii) the similarity of generated apps to mock-up artifacts (both visually and structurally), and (iii) the potential industrial applicability of our system, through semi-structured interviews

with mobile designers and developers at Google, Huawei and Facebook. Our results show that our CNN-based GUI-component classifier achieves a top-1 average precision of 91% (*i.e.*, when the top class predicted by the CNN is correct), our generated applications share high visual similarity to their mock-up artifacts, the code structure for generated apps is similar to that of real applications, and REDRAW has the potential to improve and facilitate the prototyping and development of mobile apps with some practical extensions. Our evaluation also illustrates how REDRAW outperforms other related approaches for mobile application prototyping, REMAUI [232] and pix2code [127]. Finally, we provide a detailed discussion of the limitations of our approach and promising avenues for future research that build upon the core ideas presented.

In summary, the project presented in this chapter makes the following noteworthy contributions:

- The introduction of a novel approach for prototyping software GUIs rooted in a combination of techniques drawn from program analysis, MSR, ML, and CV; and an implementation of this approach in a tool called REDRAW for the Android platform;
- A comprehensive empirical evaluation of REDRAW, measuring several complimentary quality metrics, offering comparison to related work, and describing feedback from industry professionals regarding its utility;
- An online appendix [80] showcasing screenshots of generated apps and study replication information;
- As part of implementing REDRAW we collected the largest known dataset of mobile application GUI data containing screenshots and GUI related metadata for over 14k screens and over 190k GUI-components.
- Publicly available open source versions of the REDRAW code, datasets, and trained ML models [80].

4.1 Background & Problem Statement

4.1.1 Convolutional Neural Network (CNN) Background

In order to help classify images of GUI-components into their domain specific types, RE-DRAW utilizes a Convolutional Neural Network (CNN). To provide background for the unfamiliar reader, in this sub-section we give an overview of a typical CNN architecture, explaining elements of the architecture that enable accurate image classification. However, for more comprehensive descriptions of CNNs, we refer readers to [186] & [182].

CNN Overview: Fig. 4.1 illustrates the basic components of a traditional CNN architecture. As with most types of artificial neural networks, CNNs typically encompass several different *layers* starting with an input layer where an image is passed into the network, then to hidden layers where abstract features, and weights representing the “importance” of features for a target task are learned. CNNs derive their name from unique “convolutional” layers which operate upon the mathematical principle of a convolution [33]. The purpose of the convolutional layers, shown in blue in Figure 4.1, are to extract features from images. Most images are stored as a three (or four) dimensional matrix of numbers, where each dimension of the matrix represents the intensity of a color channel (*e.g.*, RGB). Convolutional layers operate upon these matrices using a *filter* (also called kernel, or feature detector), which can be thought of as a sliding window of size n by m that slides across an set of matrices representing an image. This window applies a convolution operation (*i.e.*, an element-wise matrix multiplication) creating a *feature map*, which represents extracted image features. As convolution layers are applied in succession, more abstract features are learned from the original image. *Max Pooling* layers also operate as a sliding window, pooling maximum values in the feature maps to reduce dimensionality. Finally, fully-connected layers and a softmax classifier act as a multi-layer perceptron to perform classification. CNN training is typically performed using gradient descent, and back-propagation of error gradients.

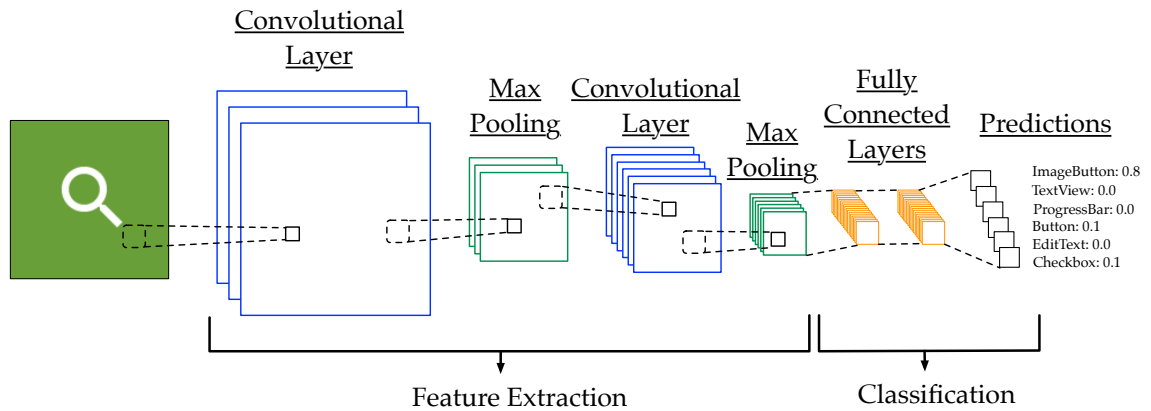


Figure 4.1: Typical Components of CNN Architecture

Convolutional Layers: Convolutional layers extract feature maps from images to learn high level features. The size of this feature map results from three parameters: (i) the number of filters used, (ii) the stride of the sliding window, and (iii) whether or not padding is applied. Leveraging multiple filters allows for multi-dimensional feature maps, the stride corresponds to the distance the sliding window moves during each iteration, and padding can be applied to learn features from the borders of an input image. These feature maps are intended to represent abstract features from images, which inform the prediction process.

Rectified Linear Units (ReLUs): Traditionally, an element of non-linearity is introduced after each convolutional layer, as the convolution operator is linear in nature, which may not correspond to non-linear nature of data being learned. The typical manner in which this non-linearity is introduced is through *Rectified Linear Units* (ReLUs). The operation these units perform is simple in nature, replacing all negative values in a feature map with zeros. After the convolutions and ReLU operations have been performed, the resulting feature map is typically subjected to *max pooling* (Fig. 4.1).

Max Pooling: Max pooling again operates as as sliding window, but instead of performing a convolution, simply pools the maximum value from each step of the sliding window. This allows for a reduction in the dimensionality of the data while extracting salient features.

Fully Connected Layers: The layers described thus far in the network have been focused on deriving features from images. Therefore, the final layers of the network must utilize

these features to compute predictions about classes for classifications. This is accomplished via the *fully connected layers*, which act as a multi-layer perceptron typically utilizing a softmax activation function.

CNN Training Procedure: Training a CNN is accomplished through back-propagation. After the initialization of all the network parameters, initial weights are set to random values. Then input images are fed through the network layers in the forward direction, and the total error across all output classes is calculated. This error is back-propagated through the network and *gradient descent* is used to calculate error gradients for the network weights which are then updated to minimize the output error. A *learning rate* controls the degree to which weights are updated based on the gradient calculations. This process is repeated over the entire training image set, which allows for training both feature extraction and classification in one automated process. After training is complete, the network should be capable of effective classification of input images.

4.1.2 Problem Definition

Given the definitions specified in Chapter 2, the problem that we aim to solve with our proposed approach is the following:

Problem Statement: *Given a mock-up artifact, generate a prototype application that closely resembles the mock-up GUI both visually, and in terms of expected structure of the GUI-hierarchy.*

As we describe in Sec. 4.2, this problem can be broken down into three distinct tasks including the *detection* and *classification* of GUI-components, and the *assembly* of a realistic GUI-hierarchy and related code. In the scope of this project, we focus on automatically generating GUIs for mobile apps that are visually and structurally similar (in terms of their GUI hierarchy). To accomplish this we investigate the ability of our proposed approach to automatically prototype applications from two types of mock-up artifacts, (i) images of existing applications, and (ii) Sketch [86] mock-ups reverse engineered from existing

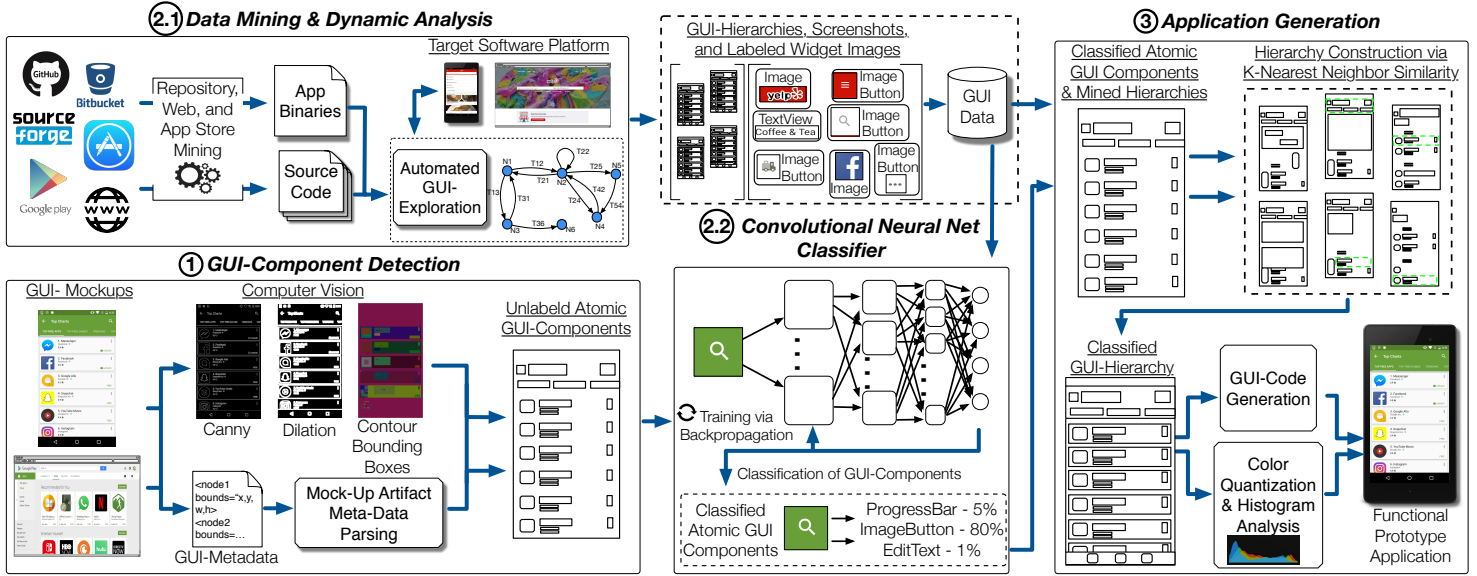


Figure 4.2: Overview of Proposed Approach for Automated GUI-Prototyping

popular applications. We utilize these types of artifacts as real mockups are typically not available for open source mobile apps and thus could not be utilized in our study. It should be noted that the two types of mock-up artifacts used in our investigation of REDRAW may not capture certain ambiguities that exist in mock-ups created during the course of a real software design process. We discuss the implications of this in Sec. 4.5.

4.2 Approach Description

We describe our approach for GUI prototyping around the three major phases of the process: *detection*, *classification*, & *assembly*. Fig. 4.2 illustrates an overview of the process that we will refer to throughout the description of the approach. At a high-level, our approach first *detects* GUI-components from a mock-up artifact by either utilizing CV techniques or parsing meta-data directly from mock-up artifacts generated using professional photo-editing software. Second, to *classify* the detected GUI-components into proper types, we propose to train a CNN using GUI data gleaned from large-scale automated dynamic analysis of applications extracted by mining software repositories. The trained CNN can then be applied to mock-up artifacts to classify detected components.

Finally, to construct a suitable GUI-hierarchy (*e.g.*, proper groupings of GUI-components in GUI-containers) we utilize a KNN-based algorithm that leverages the GUI-information extracted from the large-scale dynamic analysis to *assemble* a realistic nested hierarchy of GUI-components and GUI-containers. To illustrate our general approach, for each phase we first describe the proposed methodology and design decisions at a high level and then discuss the implementation details specific to our instantiation of REDRAW for the Android platform.

4.2.1 Phase 1 - Detection of GUI-Components

The first task required of a GUI-prototyping approach is detecting the GUI-components that exist in a mock-up artifact. The main *goal* of this phase is to accurately infer the bounding boxes of atomic GUI-component elements (in terms of pixel-based coordinates) from a mock-up artifact. This allows individual images of GUI-components to be cropped and extracted in order to be utilized in the later stages of the prototyping process. This phase can be accomplished via one of two methodologies: (i) parsing data from mock-up artifacts, or (ii) using CV techniques to detect GUI-components. A visualization of this phase is illustrated in Fig. 4.2-①. In the following subsections we describe the *detection* procedure for both of these methodologies as well as our specific implementation within REDRAW.

4.2.1.1 Parsing Data from Design Mockups

The first method for detecting the GUI-components that exist in a mock-up artifact, shown in the bottom portion of Fig. 4.2-①, is to utilize the information encoded into mock-up artifacts. Given the importance of UI/UX in today’s consumer facing software, many designers and small teams of developers work with professional grade image editing software, such as Photoshop [4] or Sketch [86] to create either wireframe or pixel perfect static images of GUIs that comprise mock-up artifacts. During this process photo-editing or design software is typically used to create a blank canvas with dimensions that match

a target device screen or display area (with some design software facilitating scaling to multiple screen sizes [4, 86]). Then, images representing GUI-components are placed as editable objects on top of this canvas to construct the mock-up. Most of these tools are capable of exporting the mock-up artifacts in formats that encode spatial information about the objects on the canvas, such as using the Scalable Vector Graphics (.svg) format or html output [60]. Information about the layouts of objects, including the bounding boxes of these objects, can be parsed from these output formats, resulting in highly accurate detection of components. Therefore, if this metadata for the mock-up artifacts is available, it can be parsed to obtain extremely accurate bounding boxes for GUI-components that exist in a mock-up artifact which can then be utilized in the remainder of the prototyping process.

Given the spatial information encoded in metadata that is sometimes available in mock-up artifacts, one may question whether this information can also be used to reconstruct a hierarchical representation of GUI-components that could later aid in the code conversion process. Unfortunately, realistic *GUI-hierarchies* typically cannot be feasibly parsed from such artifacts for at least the following two reasons: (i) designers using photo-editing software to create mock-ups tend to encode a different hierarchal structure than a developer would, due to a designer lacking knowledge regarding the best manner in which to programmatically arrange GUI-components on a screen [224]; (ii) limitations in photo-editing software can prohibit the creation of programmatically proper spatial layouts. Thus, any hierarchical structure parsed out of such artifacts is likely to be specific to designers' preferences, or restricted based on the capabilities of photo-editing software, limiting applicability in our prototyping scenario. For example, a designer might not provide enough GUI-containers to create an effective reactive mobile layout, or photo-editing software might not allow for relative positioning of GUI-components that scale across different screen sizes.

4.2.1.2 Using CV Techniques for GUI-component Detection:

While parsing information from mock-ups results in highly accurate bounding boxes for GUI-components this info may not always be available, either due to limitations in the photo-editing software being used or differing design practices, such as digitally or physically sketching mockups using pen displays, tablets, or paper. In these cases, a mock-up artifact may consist only of an image, and thus CV techniques are needed to identify relevant GUI-component info. To support these scenarios, our approach builds upon the CV techniques from [232] to detect GUI-component bounding boxes. This process uses a series of different CV techniques (Fig. 4.2(1)) to infer bounding boxes around objects corresponding to GUI components in an image. First, Canny’s edge detection algorithm [135] is used to detect the edges of objects in an image. Then these edges are dilated to merge edges close to one another. Finally, the contours of those edges are used to derive bounding boxes around atomic GUI-components. Other heuristics for merging text-based components using Optical Character Recognition (OCR) are used to merge the bounding boxes of logical blocks of text (*e.g.*, rather than detecting each word as its own component, sentences and paragraphs of text are merged).

4.2.1.3 ReDraw Implementation - GUI Component Detection

In implementing REDRAW, to support the scenario where metadata can be gleaned from mock-ups for Android applications we target artifacts created using the Marketch [60] plugin for Sketch [86], which exports mock-ups as a combination of `html` & `javascript`. Sketch is popular among mobile developers and offers extensive customization through a large library of plugins [87]. REDRAW parses the bounding boxes of GUI-components contained within the exported Marketch files.

To support the scenario where meta-data related to mock-ups is not available, REDRAW uses CV techniques to automatically infer the bounding boxes of components from a static image. To accomplish this, we re-implemented the approach described in [232].

Thus, the input to the GUI-component detection phase of REDRAW is either a screenshot and corresponding `marketch` file (to which the marketch parsing procedure is applied), or a single screenshot (to which CV-based techniques are applied). The end result of the GUI-component detection process is a set of bounding box coordinates situated within the original input screenshot and a collection of images cropped from the original screenshot according to the derived bounding boxes that depict atomic GUI-components. This information is later fed into a CNN to be classified into Android specific component types in Phase 2.2. It should be noted that only *GUI-components* are detected during this process. On the other hand *GUI-containers* and the corresponding GUI-hierarchy are constructed in the *assembly* phase described in Sec. [4.2.3](#).

4.2.2 Phase 2 - GUI-component Classification

Once the bounding boxes of atomic GUI-component elements have been *detected* from a mock-up artifact, the next step in the prototyping process is to *classify* cropped images of specific GUI components into their domain specific types. To do this, we propose a data-driven and ML-based approach that utilizes CNNs. As illustrated in Fig. [4.2](#)([2.1](#)) and Fig. [4.2](#)([2.2](#)), this phase has two major parts: (i) large scale software repository mining and automated dynamic analysis, and (ii) the training and application of a CNN to classify images of GUI-components. In the following subsections we first discuss the motivation and implementation of the repository mining and dynamic analysis processes before discussing the rationale for using a CNN and our specific architecture and implementation within REDRAW.

4.2.2.1 Phase 2.1 - Large-Scale Software Repository Mining and Dynamic Analysis

Given their supervised nature and deep architectures, CNNs aimed at the image classification task require a large amount of training data to achieve precise classification. Training data for traditional CNN image classification networks typically consists of a large set of

images labeled with their corresponding classes, where labels correspond to the primary subject in the image. Traditionally, such datasets have to be manually procured, wherein humans painstakingly label each image in the dataset. However, we propose a methodology that *automates* the creation of labeled training data consisting of images of specific GUI-components cropped from full screenshots and labels corresponding to their domain specific type (*e.g.*, Buttons, or Spinners in Android) using fully-automated dynamic program analysis.

Our key insight for this automated dynamic analysis process is the following: *during automated exploration of software mined from large repositories, platform specific frameworks can be utilized to extract meta-data describing the GUI, which can then be transformed into a large labeled training set suitable for a CNN.* As illustrated in Fig. [4.2](#)(2.1), this process can be automated by mining software repositories to extract executables. Then a wealth of research in automated input generation for GUI-based testing of applications can be used to automatically execute mined apps by simulating user-input. For instance, if the target is a mobile app, input generation techniques relying on random-based [\[208, 13, 53, 249, 269\]](#), systematic [\[124, 122, 121, 219, 227\]](#), model-based [\[121, 268, 124, 141, 166, 273, 203\]](#), or evolutionary [\[214, 212\]](#) strategies could be adopted for this task. As the app is executed, screenshots and GUI-related metadata can be automatically extracted for each unique observed screen or layout of an app. Other similar automated GUI-ripping or crawling approaches can also be adapted for other platforms such as the web [\[243, 258, 244, 143, 234\]](#).

Screenshots can be captured using third party software or utilities included with a target operating system. GUI-related metadata can be collected from a variety of sources including accessibility services [\[160\]](#), html DOM information, or UI-frameworks such as `uiautomator` [\[14\]](#). The GUI-metadata and screenshots can then be used to extract sub-images of GUI-components with their labeled types parsed from the related metadata describing each screen. The underlying quality of the resulting dataset relates to how well the labels describe the type of GUI-components displayed on a screen. Given that many of the software UI-frameworks that would be utilized to mine such data pull their

information directly from utilities that render application GUI-components on the screen, this information is likely to be highly accurate. However, there are certain situations where the information gleaned from these frameworks contains minor inaccuracies or irrelevant cases. We discuss these cases and steps that can be taken to mitigate them in Sec. [4.2.2.4](#).

4.2.2.2 ReDraw Implementation - Software Repository Mining and Automated Dynamic Analysis

To procure a large set of Android apps to construct our training, validation, and test corpora for our CNN we mined free apps from Google Play at scale. To ensure the representativeness and quality of the apps mined, we extracted all categories from the Google Play store as of June 2017. Then we filtered out any category that primarily consisted of games, as games tend to use non-standard types of GUI-components that cannot be automatically extracted. This left us with a total of 39 categories. We then used a Google Play API library [\[46\]](#) to download the top 240 APKs from each category, excluding duplicates that existed in more than one category. This resulted in a total of 8,878 unique APKs after accounting for duplicates cross-listed across categories.

To extract information from the mined APKs, we implemented a large-scale dynamic analysis engine, called the *Execution Engine* that utilizes a systematic automated input generation approach based on our prior work on CRASHSCOPE and MONKEYLAB [\[203\]](#), [\[225\]](#), [\[219\]](#), [\[227\]](#) to explore the apps and extract screenshots and GUI-related information for visited screens. More specifically, our systematic GUI-exploration navigates a target app's GUI in a Depth-First-Search (DFS) manner to exercise tappable, long-tappable, and typeable (*e.g.*, capable of accepting text input) components. During the systematic exploration we used Android's `uiautomator` framework [\[14\]](#) to extract GUI-related info as `xml` files that describe the hierarchy and various properties of components displayed on a given screen. We used the Android `screencap` utility to collect screenshots. The `uiautomator xml` files contain various attributes and properties of each GUI-component displayed on an Android application screen, including the bounding boxes (*e.g.*, precise location and area within the

screen) and component types (*e.g.*, EditText, Toggle Button). These attributes allow for individual sub-images for each GUI-component displayed on a given screen to be extracted from the corresponding screenshot and automatically labeled with their proper type.

The implementation of our DFS exploration strategy utilizes a state machine model where states are considered unique app screens, as indicated by their activity name and displayed window (*e.g.*, dialog box) extracted using the `adb shell dumpsys window` command. To allow for feasible execution times across the more than 8.8k apps in our dataset while still exploring several app screens, we limited our exploration strategy to exercising 50 actions per app. Prior studies have shown that most automated input generation approaches for Android tend to reach near-peak coverage (*e.g.*, between ≈ 20 and 40% statement coverage) after 5 minutes of exploration [142]. While different input generation approaches tend to exhibit different numbers of actions per given unit of time, our past work shows that our automated input generation approach achieves competitive coverage to similar approaches [219], and our stipulation of 50 actions comfortably exceeds 5 minutes per app. Furthermore, our goal with this large scale analysis was not to completely explore each application, but rather ensure a diverse set of screens and GUI-Component types. For each app the *Execution Engine* extracted `uiautomator` files and screenshot pairs for the top six unique screens of each app based on the number of times the screen was visited. If fewer than six screens were collected for a given app, then the information for all screens was collected. Our large scale *Execution Engine* operates in a parallel fashion, where a centralized dispatcher allocated jobs to workers, where each worker is connected to one physical Nexus 7 tablet and is responsible for coordinating the execution of incoming jobs. During the dynamic analysis process, each job consists of the systematic execution of a single app from our dataset. When a worker finished with a job, it then notified the dispatcher which in turn allocates a new job. This process proceeded in parallel across 5 workers until all applications in our dataset had been explored. Since Ads are popular in free apps [245, 163], and are typically made up of dynamic *WebViews* and not native

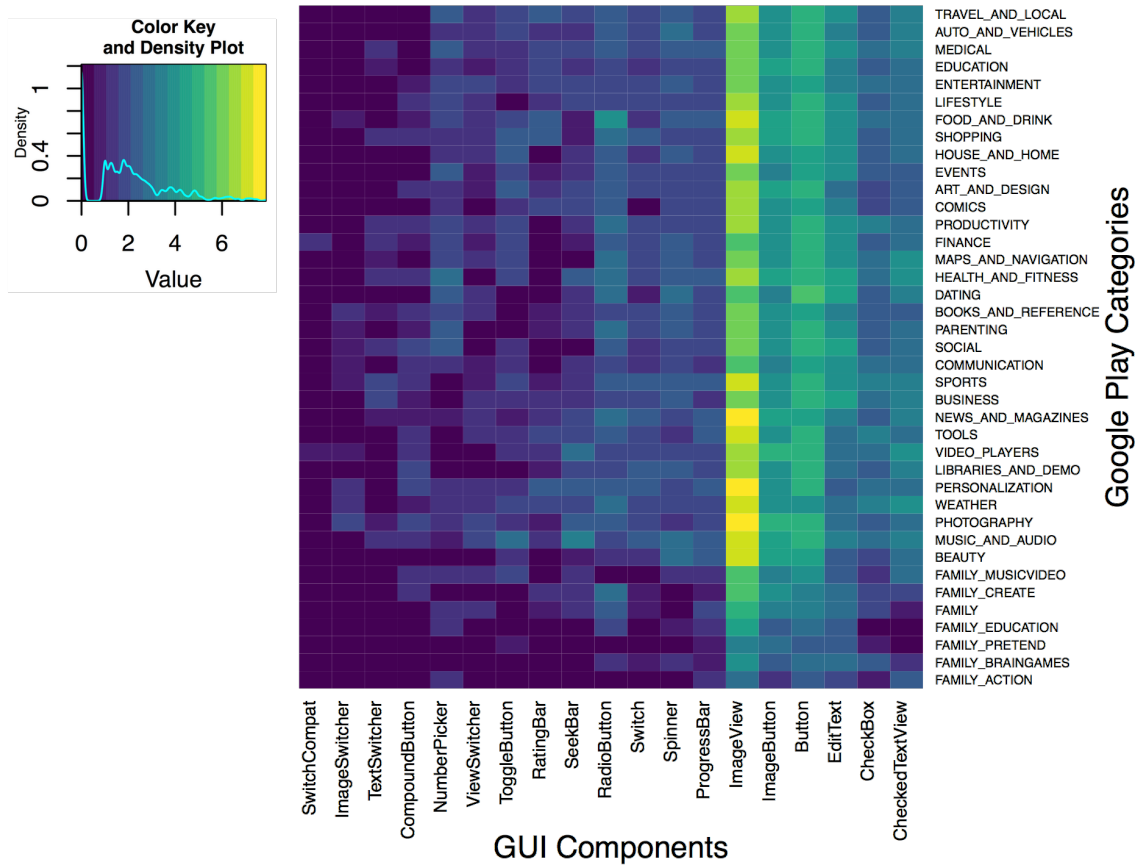


Figure 4.3: Heat-map of GUI Components by Category

components, we used *Xposed* [241] to block Ads in apps that might otherwise obscure other types of native components.

This process resulted in a dataset of GUI-information and screenshots for 19,786 unique app screens containing over 431,747 native Android GUI-components and containers which, to the best of the authors knowledge, is one of the largest such datasets collected to date behind the RICO dataset [151]. In Fig. 4.3 we illustrate the frequency in logarithmic-scale of the top-19 observed components by app category using a heat-map based on the frequency of components appearing from apps within a particular category (excluding *TextViews* as they are, unsurprisingly, the most popular type of component observed, comprising $\approx 25\%$ of components). The distributions of components in this dataset illustrate two major points. First, while *ImageViews* and *TextViews* tend to comprise a large number of

the components observed in practice, developers also heavily rely on other types of native Android components to implement key pieces of app functionality. For instance, `Buttons`, `CheckedTextViews`, and `RadioButtons` combined were used over 20k times across the apps in our dataset. Second, we observed certain types of components may be more popular for different categories of apps. For instance, apps from the category of "MUSIC_AND_AUDIO" tend to make much higher use of *SeekBar* and *ToggleButton* components to implement the expected functionalities of a media player, such as scrubbing through music and video files. These findings illustrate that for an approach to be able to effectively generate prototypes for a diverse set of mobile apps, it must be capable of correctly detecting and classifying popular types of GUI-components to support varying functionality.

4.2.2.3 Phase 2.2 - CNN Classification of GUI-Components

Once the labeled training data set has been collected, we need to train a ML approach to extract salient features from the GUI-component images, and classify incoming images based upon these extracted features. To accomplish this our approach leverages recent advances in CNNs. The main advantage of CNNs over other image classification approaches is that the architecture allows for automated extraction of abstract features from image data, approximation of non-linear relationships, application of the principle of data-locality, and classification in an end-to-end trainable architecture.

4.2.2.4 ReDraw Implementation - CNN Classifier

Once the GUI-components in a target mock-up artifact have been detected using either mock-up meta-data or CV-based techniques, REDRAW must effectively classify these components. To accomplish this REDRAW implements a CNN capable of classifying a target image of a GUI-component into one of the 15 most-popular types of components observed in our dataset. In this subsection, we first describe the data-cleaning process used to generate the training, validation, and test datasets (examples of which are shown in Fig. [4.4](#)) before describing our CNN architecture and the training procedure we employ.

Data Cleaning: We implemented several types of preprocessing and filtering techniques to help reduce noise. More specifically, we implemented filtering processes at three differing levels of granularity: (i) application, (ii) screen & (iii) GUI-component level.

While future versions of REDRAW may support non-native apps, to provide an appropriate scope for rigorous experimentation, we have implemented REDRAW with support for prototyping *native* Android applications. Thus, once we collected the `xml` and screenshot files, it is important to apply filters in order to discard applications that are non-native, including games and hybrid applications. Thus, we applied the following app-level filtering methodologies:

- ***Hybrid Applications:*** We filtered applications that utilize Apache Cordova [18] to implement mobile apps using web-technologies such as `html` and `CSS`. To accomplish this we first decompiled the APKs using `Apktool` [19] to get the resources used in the application. We then discarded the applications that contained a `www` folder with `html` code inside.
- ***Non-Standard GUI Frameworks:*** Some modern apps utilize third party graphical frameworks or libraries to create highly-customized GUIs. While such frameworks tend to be used heavily for creating mobile games, they can also be used to create UIs for more traditional applications. One such popular framework is the Unity [103] game engine. Thus, to avoid applications that utilize this engine we filtered out applications that contain the folder structure `com/unity3d/player` inside the code folder after decompilation with `Apktool`.

This process resulted in the removal of 223 applications and a dataset consisting of 8,655 apps to which we then applied screen-level filtering. At the Screen-level, we implemented the following pre-processing techniques:

- ***Filtering out Landscape screens:*** To keep the height and width of all screens consistent, we only collected data from screens that displayed in the portrait orien-

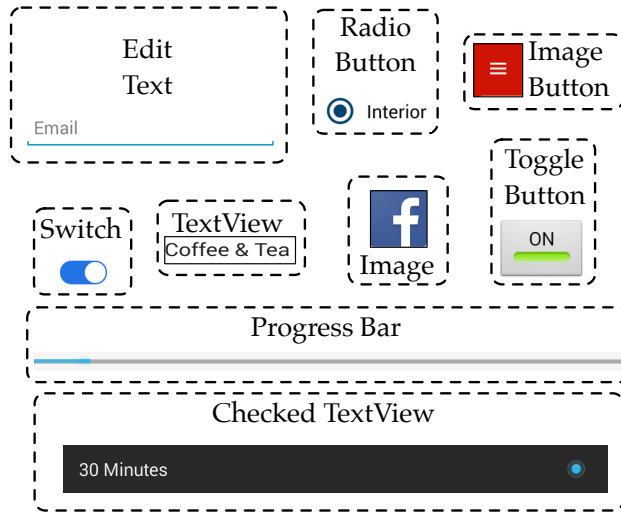


Figure 4.4: Example of a subset of ReDraw’s training data set consisting of GUI-Component sub-images and domain (Android) specific labels. Images and corresponding Labels are grouped according to the dashed-lines.

tation. Thus, we checked the size of the extracted screenshots and verified that the width and the height correspond to 1200x1920, the landscape oriented screen size used on our target Nexus 7 devices. However, there are some corner cases in which the images had the correct portrait size but it was on landscape. So, to overcome this we checked the extracted `uiautomator xml` file and validated the size of the screen to ensure a portrait orientation.

- **Filtering Screens containing only Layout components:** In Android, Layout components are used as containers that group together other types of functional components such as *Buttons* and *Spinners*. However, some screens may consist only of layout components. Thus to ensure variety in our dataset, we analyzed the `uiautomator xml` files extracted during dynamic analysis to discard screens that are only comprised of Layout components such as *LinearLayout*, *GridLayout*, and *FrameLayout* among others.
- **Filtering WebViews:** While many of the most popular Android apps are native, some apps may be hybrid in nature, that is utilizing web content within a native app wrapper. Because such apps use components that cannot be extracted via

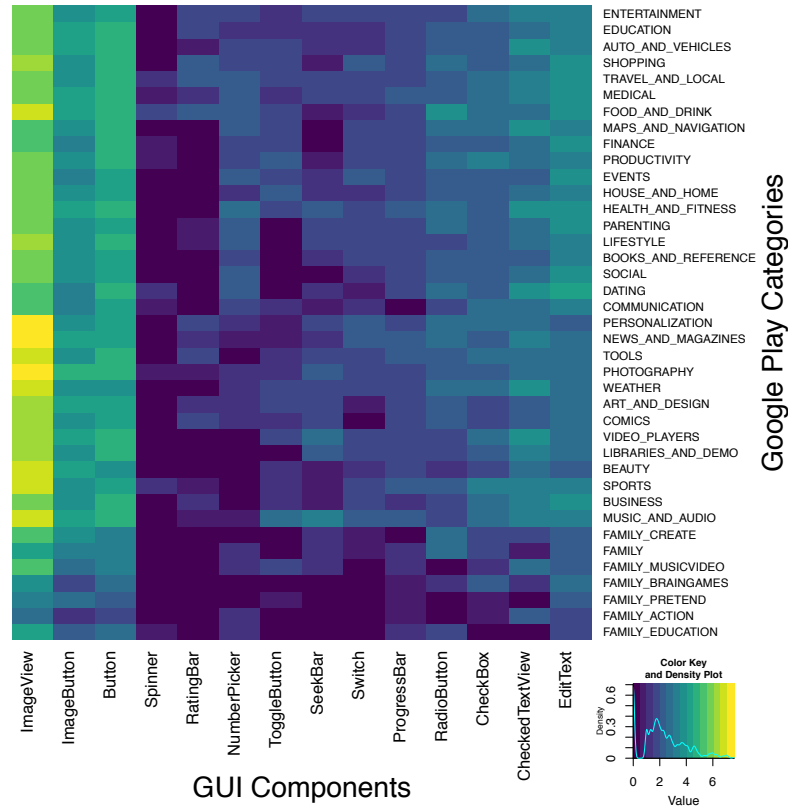


Figure 4.5: Heat-map of GUI Components by Category After Filtering

uiautomator we discard them from our dataset by removing screens where a `WebView` occupied more than 50% of the screen area.

After these filtering techniques were applied, 2,129 applications and 4,954 screens were removed, and the resulting dataset contained 14,382 unique screens with 431,747 unique components from 6,538 applications. We used the information in the `uiautomator.xml` files to extract the bounding boxes of *leaf-level* GUI-components in the GUI-hierarchies. We only extract leaf-level components in order to align our dataset with components detected from mock-ups. Intuitively it is unlikely that container components (*e.g.*, non-leaf nodes) would exhibit significant distinguishable features that a ML approach would be able to derive in order to perform accurate classification (hence, the use of our KNN-based approach is described in Sec. [4.2.3](#)). Furthermore, it is unclear how such a GUI-container

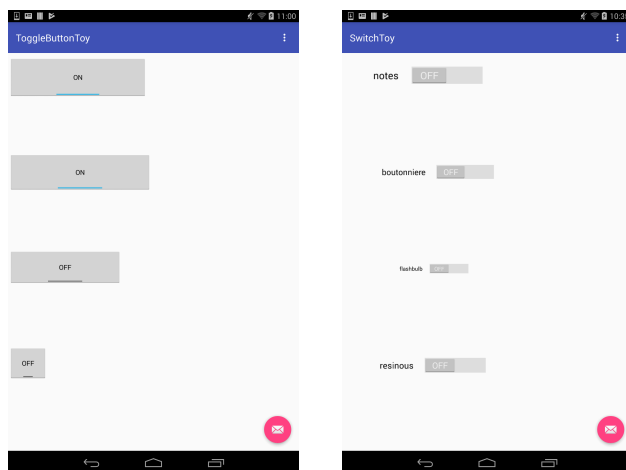


Figure 4.6: Screenshots of synthetically generated applications containing toggle buttons and switches

classification network would be used to iteratively build a GUI-structure. We performed a final filtering of the extracted leaf components:

- **Filtering Noise:** We observed that in rare cases the bounds of components would not be valid (*e.g.*, extending beyond the borders of the screen, or represented as zero or negative areas) or components would not have a type assigned to them. Thus, we filter out these cases.
- **Filtering Solid Colors:** We also observed that in certain circumstances, extracted components were made up of a single solid color, or in rarer cases two solid colors. This typically occurred due to instances where the view hierarchy of a screen had loaded, but the content was still rendering on the page or being loaded over the network, when a screenshot was captured. Thus, we discarded such cases.
- **Filtering Rare GUI-Components:** In our dataset we found that some components only appeared very few times, therefore, we filtered out any component with less than 200 instances in the initial dataset, leading to 15 GUI-component types in our dataset.

The data-cleaning process described above resulted in the removal of 240,447 components resulting in 191,300 labeled images of GUI-components from 6,538 applications. We provide a heat-map illustrating the popularity of components across apps from different Google Play categories in Fig. 4.5. To ensure the integrity of our dataset, we randomly sampled a statistically significant sample of 1,000 GUI-component images (corresponding to confidence interval of ± 3.09 at a 95% confidence level), and had one author manually inspect all 1,000 images and labels to ensure the dataset integrity.

Data Augmentation: Before segmenting the resulting data into training, test, and validation sets, we followed procedures from previous work [186] and applied data augmentation techniques to increase the size of our dataset in order to ensure proper training support for underrepresented classes and help to combat overfitting to the training set. Like many datasets procured using “naturally” occurring data, our dataset suffers from imbalanced classes. That is, the number of labeled images in our training set are skewed toward certain classes, resulting in certain classes that have high support, and others that have low support. Thus, to balance our dataset, we performed two types of data augmentation: *synthetic app generation* and *color perturbation*. For the sake of clarity, we will refer to data collected using our automated dynamic analysis approach as *organic data* (*i.e.*, the data extracted from Google Play) and data generated via synthetic means as *synthetic data* (*i.e.*, generated either via synthetic app generation or color perturbation).

To generate synthetic data for underrepresented components, we implemented an *app synthesizer* capable of generating Android apps consisting of only underrepresented components. The app synthesizer is a Java application that is capable of automatically generating single-screen Android applications containing four instances of GUI-components (with randomized attributes) for 12 GUI-component classes in our dataset that had less than 10K observable instances. The synthesizer places the four GUI-components of the specified type on a single app screen with randomized sizes and values (*e.g.*, numbers for a number picker, size and state for a toggle button). Two screenshots of synthesized applications used to augment the Toggle button and Switch classes are illustrated in Fig. 4.6. We

ran these apps through our *Execution Engine*, collecting the `uiautomator xml` files and screenshots from the single generated screen for each app. After the screenshots and `uiautomator` files were collected, we extracted *only* the target underrepresented components from each screenshot (note that in Fig. 4.6 there is a header title and button generated when creating a standard Android app), all other component types are ignored. 250 apps for each underrepresented GUI-component were synthesized, resulting in creating an extra 1K components for each class and 12K total additional GUI-components.

While our application generator helps to rectify the imbalanced class support to an extent, it does not completely balance our classes and may be prone to overfitting. Thus, to ensure proper support across all classes and to combat overfitting, we follow the guidance outlined in related work [186] to perform *color perturbation* on both the organic and synthetic images in our dataset. More specifically, our color perturbation procedure extracts the RGB values for each pixel in an input image and converts the values to the HSB (Hue, Saturation, Brightness) color space. The HSB color space represents colors as part of a cylindrical or cone model where color hues are represented by degrees. Thus, to shift the colors of a target image, our perturbation approach randomly chooses a degree value by which each pixel in the image is shifted. This ensures that color hues that were the same in the original image, all shift to the same new color hue in the perturbed image, preserving the visual coherency of the perturbed images. We applied color perturbation to the training set of images until each class of GUI-component had at least 5K labeled instances, as described below.

Data Segmentation: We created a the training, validation, and test datasets for our CNN such that the training dataset contained both *organic* and *synthetic* data, but the test and validation datasets contained **only** *organic* data, unseen in the training phase of the CNN. To accomplish this, we randomly segmented our dataset of *organic* components extracted from Google Play into *training* (75%), *validation* (15%), and *test* (10%) sets. Then for the training set, we added the synthetically generated components to the set of organic GUI-component training images, and performed color perturbation on **only** the

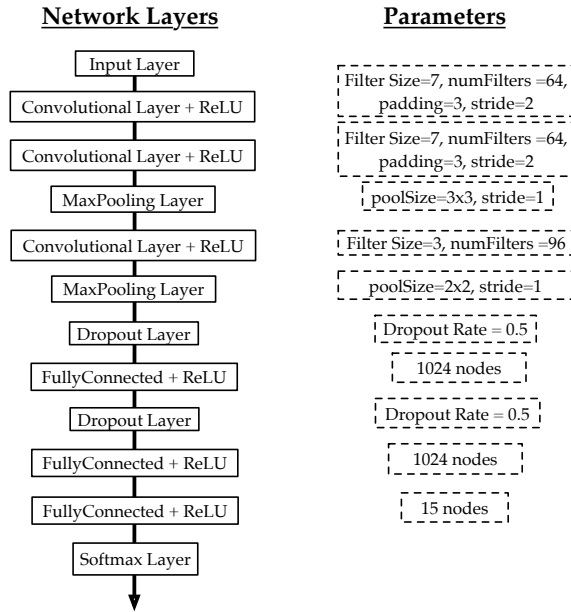


Figure 4.7: REDRAW CNN Architecture

training data (after segmentation) until each class had at least 5K training examples. Thus, the training set contained *both* organic and synthetically generated data, and the validation and test sets contained *only* organic data. This segmentation methodology closely follows prior work on CNNs [186].

ReDraw’s CNN Architecture: Our CNN architecture is illustrated in Fig. 4.7. Our network uses an architecture similar to that of AlexNet [186], with two less convolutional layers (3 instead of 5), and is implemented in MATLAB using the Neural Network [62], Parallel Computing [96], and Computer Vision [95] toolkits. While “deeper” architectures do exist [275, 257, 167] and have been shown to achieve better performance on large-scale image recognition benchmarks, this comes at the cost of dramatically longer training times and a larger set of parameters to tune. Since our goal is to classify 15 classes of the most popular Android GUI-components, we do not need the capacity of deeper networks aiming to classify thousands of image categories. We leave deeper architectures and larger numbers of image categories as future work. Also, this allowed our CNN to converge in a matter of hours rather than weeks, and as we illustrate, still achieve high precision.

To tune our CNN, we performed small scale experiments by randomly sampling 1K images from each class to build a small training/validation/test set (75%, 15%, 10%) for faster training times (Note, these datasets are separate from the full set used to train/validate/test the network described earlier). During these experiments we iteratively recorded the accuracy on our validation set, and recorded the final accuracy on the test set. We tuned the location of layers and parameters of the network until we achieved peak test accuracy with our randomly sampled dataset.

Training the CNN: To train REDRAW’s network we utilized our derived training set; we trained our CNN end-to-end using back-propagation and stochastic gradient descent with momentum (SGDM), in conjunction with a technique to prevent our network from overfitting to our training data. That is, every five epochs (*e.g.*, entire training set passing through the network once) we test the accuracy of our CNN on the validation set, saving a copy of the learned weights of the classifier at the same time. If we observe our accuracy decrease for more than two checkpoints, we terminate the training procedure. We varied our learning rate from 0.001 to 1×10^{-5} after 50 epochs, and then dropped the rate again to 1×10^{-6} after 75 epochs until training terminated. Gradually decreasing the learning rate allows for the network to “fine-tune” the learned weights over time, leading to an increase in overall classification precision [186]. Our network training time was 17 hours, 12 minutes on a machine with a single Nvidia Tesla K40 GPU.

Using the CNN for Classification: Once the CNN has been trained, new, unseen images can be fed into the network resulting in a series of classification scores corresponding to each class. In the case of ReDraw, the component class with the highest confidence is assigned to be the label for a given target image. We present an evaluation of the classification accuracy of REDRAW’s CNN using the dataset described in this subsection later in Sec. 4.3 & 4.4.

Algorithm 1: KNN Container Determination

```
Input: InputNodes // Either leaf components or other containers
Output: Containers // Groupings of input components
1 while canGroupMoreNodes() // While groupings exist
2 do
    // For each screen in the mined data
3 foreach Screen  $S \in$  Dataset do
4      $TargetNodes = S.getTargetNodes()$   $score = \frac{TargetNodes() \cap InputNodes}{TargetNodes() \cup InputNodes}$  // IOU
5     if  $score > curmax$  then
6          $curmax = score$ 
7          $MatchedScreen = S$ 
8     end
9 end
10  $TargetNodes = MatchedScreen.getTargetNodes()$ 
     $InputNodes.remove(TargetNodes \cap InputNodes)$   $Containers.addContainers(MatchedScreen)$ 
11 end
```

4.2.3 Phase 3 - Application Assembly

The final task of the prototyping process is to assemble app GUI code, which involves three phases (Fig. 4.2-3): (i) building a proper hierarchy of components and containers, (ii) inferring stylistic details from a target mock-up artifact, and (iii) assembling the app.

4.2.3.1 Deriving GUI-Hierarchies

In order to infer a realistic hierarchy from the classified set of components, our approach utilizes a KNN technique (Alg. 1) for constructing the GUI hierarchy. This algorithm takes the set of detected and classified GUI-components represented as nodes in a single level tree (*InputNodes*) as input. Then, for each screen in our dataset collected from automated dynamic analysis, Alg. 1 first extracts a set of *TargetNodes* that correspond the hierarchy level of the *InputNodes* (Alg. 1-line 4), which are leaf nodes for the first pass of the algorithm. Next, the *InputNodes* are compared to each set of extracted (*TargetNodes*) using a similarity metric based on the intersection over union (IOU) of screen area occupied by the bounding boxes of overlapping components (Alg. 1-line 5). A matching screen is selected by taking the screen with the highest combined IOU score between the *InputNodes* and *TargetNodes*. Then, the parent container components from the components in the matched screen are selected as parent components to the matched

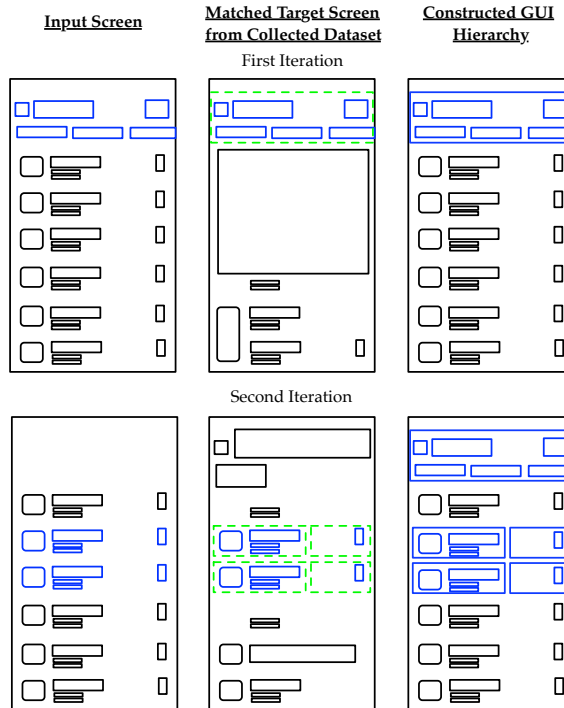


Figure 4.8: Illustration of KNN Hierarchy Construction

InputNodes. The matched *InputNodes* are then removed from the set, and the algorithm proceeds to match the remaining *InputNodes* that were not matched during the previous iteration. This procedure is applied iteratively (including grouping containers in other containers) until a specified number of levels in the hierarchy are built or all nodes have been grouped. An illustration of this algorithm is given in Figure 4.8, where matched components are highlighted in blue and containers are represented as green boxes.

It should be noted that all attributes of a component container are inherited during the hierarchy construction, including their type (e.g., `LinearLayout`, `RelativeLayout`). We can specify the number of component levels to ensure that hierarchies do not grow so large such that they would cause rendering delays on a device. The result of this process is a hierarchy built according to its similarity to existing GUI-hierarchies observed in data. Given different types of containers may behave differently, this technique has the advantage that, in addition to leaf level GUI-components being properly classified by

Listing 4.1: ReDraw’s Skeleton Main Activity Class

```
1 public class MainActivity extends Activity {
2     @Override
3     protected void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         setContentView(R.layout.main_activity);
6     }
7 }
```

Listing 4.2: Snippet from layout.xml file generated by ReDraw for the Yelp Application

```
1 <LinearLayout android:id="@+id/LinearLayout452" android:layout_height="
  127.80186dp" android:layout_marginStart="0.0dp"
  android:layout_marginTop="0.0dp" android:layout_width="400.74304dp"
  android:orientation="vertical" android:text="" android:textSize="8pt
  ">
2 <Button android:id="@+id/Button454" android:layout_height="58.45201
  dp" android:layout_marginStart="0.0dp" android:layout_marginTop=
  "0.0dp" android:layout_width="400.74304dp" android:text="Sign up
  with Google" android:textSize="8pt" style="@style/Style65"/>
3 <Button android:id="@+id/Button453" android:layout_height="50.526318
  dp" android:layout_marginStart="3.4674923dp"
  android:layout_marginTop="18.82353dp" android:layout_width="
  393.31268dp" android:text="Sign up with Facebook"
  android:textSize="8pt" style="@style/Style66"/>
4 </LinearLayout>
```

the CNN, proper types of container components are built into the GUI-hierarchy via this KNN-based approach.

4.2.3.2 Inferring Styles and Assembling a Target App

To infer stylistic details from the mock-up, our approach employs the CV techniques of Color Quantization (CQ), and Color Histogram Analysis (CHA). For GUI-components whose type does not suggest that they are displaying an image, our approach quantizes the color values of each pixel and constructs a color histogram. The most popular color values can then be used to inform style attributes of components when code is generated. For example, for a component displaying text, the most prevalent color can be used as a background and the second most prevalent color can be used for the font.

Listing 4.3: Snippet from `style.xml` file generated by ReDraw for the Yelp Application

```
1 <style name="Style63" parent="AppTheme">
2   <item name="android:textColor">#FEFEFF</item>
3 </style>
4 <style name="Style64" parent="AppTheme">
5   <item name="android:textColor">#FEFEFF</item>
6 </style>
7 <style name="Style65" parent="AppTheme">
8   <item name="android:background">#DD4B39</item>
9   <item name="android:textColor">#FEFEFF</item>
10 </style>
```

4.2.3.3 ReDraw Implementation - App Assembly

ReDraw assembles Android applications, using the KNN approach for GUI-hierarchy construction (see Sec. 4.2.3.1) and CV-based detection of color styles. The input to Alg. 1 is the set of classified "leaf-node" components from the CNN, and the output is a GUI-hierarchy. To provide sufficient data for the KNN-algorithm, a corpus including all of the info from the "cleaned" screens of the GUI-hierarchies mined from our large scale dynamic analysis process is constructed. This corpus forms the dataset *TargetNodes* to which the *InputNode* components are matched against during hierarchy construction. The GUI-hierarchy generated by the KNN for the target "leaf-node" components is then used to infer stylistic details from the original mock-up artifact. More specifically, for each component and container, we perform CQ and CHA to extract the dominant colors for each component. For components which have a text element, we apply optical character recognition (OCR) using the open source Tesseract [94] library on the original screenshot to obtain the strings.

Currently, our approach is able to infer three major types of stylistic detail from target components:

- **Background Color:** To infer the background color of components and containers, ReDraw simply utilizes the dominant color in the CHA for a specific component as the background color.

- **Font Color:** To infer the font color for components, ReDraw uses the dominant color in the CHA as the background text and the second most dominant color as the font color.
- **Font Size:** ReDraw is able to infer the font size of textual components by using the pixel based height of the bounding boxes of text-related components.

These techniques are used for both variants of the REDRAW approach (*e.g.*, mock-up based and CV based). There is ample opportunity for future work to improve upon the inference of stylistic details, particularly from mock-up artifacts. More specifically, future work could expand this process to further adapt the style of “standard” components to match stylistic details observed in a mock-up artifact. Depending upon the export format for a mock-up, ReDraw could also potentially infer additional styles such as the font utilized or properties of component shapes (*e.g.*, button bevels). While REDRAW’s current capabilities for inferring stylistic details are limited to the above three categories, in Section [4.4](#) we illustrate that these are sufficient to enable REDRAW to generate highly visually similar applications in comparison to target images.

REDRAW encodes the information regarding the GUI-hierarchy, stylistic details, and strings detected using OCR into an intermediate representation (IR) before translating it into code. This IR follows the format of `uiautomator xml` files that describes dynamic information from an Android screen. Thus, after REDRAW encodes the GUI information into the `uiautomator`-based IR, it then generates the necessary resource `xml` files (*e.g.*, files in the `res` folder of an Android app project directory) by parsing the `uiautomator`-based IR `xml` file. This process generates the following two types of resource code for the generated app: (i) the `layout.xml` code describing the general GUI structure complete with strings detected via OCR; and (ii) a `style.xml` file that stipulates the color and style information for each component gleaned via the CV techniques, and ReDraw generates the `xml` source files following the best practices stipulated in the Android developer guidelines [\[12\]](#), such as utilizing relative positioning, and proper padding and margins. In addition

to these resource `xml` files REDRAW also generates a skeleton Java class encompassing the `MainActivity` which renders the GUI stipulated in the resource `xml` files, as well as other various files required to build and package the code into an `apk`. The Skeleton `MainActivity` Java class is shown in Listing 4.1 and snippets from generated `layout.xml` & `style.xml` files for a screen from the Yelp application are shown in Listings 4.2 & 4.3. The `layout.xml` snippet of code generated by ReDraw illustrates the use of margins and relative `dp` values to stipulate the spatial properties of GUI-containers and GUI-components and references the `style.xml` file to stipulate color information. Listing 4.3 illustrates the corresponding styles and colors referenced by the `layout.xml` file.

4.3 Empirical Study Design

The *goal* of our empirical study is to evaluate REDRAW in terms of (i) the accuracy of the CNN GUI-component classifier, (ii) the similarity of the generated GUI-hierarchies to real hierarchies constructed by developers, (iii) the visual similarity of generated apps compared to mock-ups, and (iv) ReDraw’s suitability in an industrial context. The *context* of this study consists of (i) a set of 191,300 labeled images of Android GUI-components extracted from 14,382 unique app screens mined from 6,538 `APKs` from the Google Play store (see Sec. 4.2.2.2 for details) to assess the accuracy of the CNN-classifier, (ii) 83 additional screens (not included in the dataset to train and test the CNN-classifier) extracted from 32 of the highest rated apps on Google Play (top-3 in each category), (iii) nine reverse engineered Sketch mockups from eight randomly selected highly rated Google Play Apps to serve as mock-up artifacts, and (iv) two additional approaches for prototyping Android applications REMAUI [232] and pix2code [127]. The *quality focus* of this study is the effectiveness of REDRAW to generate prototype apps that are both visually similar to target mock-up artifacts, with GUI-hierarchies similar to those created by developers. To aid in achieving the goals of our study we formulated the following RQs:

- **RQ₁**: *How accurate is the CNN-based image classifier for classifying Android GUI-components?*
- **RQ₂**: *How similar are GUI-hierarchies constructed using REDRAW’s KNN algorithm compared to real GUI-hierarchies?*
- **RQ₃**: *Are the prototype applications that REDRAW generates visually similar to mock-up artifacts?*
- **RQ₄**: *Would actual mobile developers and designers consider using REDRAW as part of their workflow?*

It should be noted that in answering RQ₂-RQ₄ we use two types of mock-up artifacts (existing application screenshots, and reverse engineered Sketch mock-ups) as a proxy for real GUI-design mock-ups, and these artifacts are not a perfect approximation. More specifically, screenshots represent a finalized GUI-design, whereas real GUI design mockups may not be complete and might include ambiguities or design parameters that are able to be properly implemented in code (*i.e.*, unavailable fonts or impractical spatial layouts). Thus, we do not claim to measure REDRAW’s performance on incomplete or “in-progress” design mock-ups. However, it was not possible to obtain actual GUI design mock-ups for our study, and our target screenshots and reverse engineered mock-ups stem from widely used applications. We discuss this point further in Sec. [4.5](#).

4.3.1 RQ₁: Effectiveness of the CNN

To answer **RQ₁**, as outlined in Sec. [4.2.2.4](#) we applied a large scale automated dynamic analysis technique and various data cleaning procedures which resulted in a total of 6,538 apps, 14,382 unique screens, and 191,300 labeled images of GUI-components. To normalize support across classes and prepare training, validation and test sets in order measure the effectiveness of our CNN we applied data augmentation, and segmentation techniques also described in detail in Sec. [4.2.2.4](#). The datasets utilized are illustrated, broken down by class, in Table [4.1](#). We trained the CNN on the training set of data, avoiding overfitting

Table 4.1: Labeled GUI-Component Image Datasets

GUI-C Type	Total # (C)	Tr (O)	Tr (O+S)	Valid	Test
TextView	99,200	74,087	74,087	15,236	9,877
ImageView	53,324	39,983	39,983	7,996	5,345
Button	16,007	12,007	12,007	2,400	1,600
ImageButton	8,693	6,521	6,521	1,306	866
EditText	5,643	4,230	5,000	846	567
CheckedTextView	3,424	2,582	5,000	505	337
CheckBox	1,650	1,238	5,000	247	165
RadioButton	1,293	970	5,000	194	129
ProgressBar	406	307	5,000	60	39
SeekBar	405	304	5,000	61	40
NumberPicker	378	283	5,000	57	38
Switch	373	280	5,000	56	37
ToggleButton	265	199	5,000	40	26
RatingBar	219	164	5,000	33	22
Spinner	20	15	5,000	3	2
Total	191,300	143,170	187,598	29,040	19,090

Abbreviations for column headings: "Total#(C)"=Total # of GUI-components in each class after cleaning; "Valid"= Validation; "Tr(O)"= Training Data (Organic Components Only); "Tr(O+S)"= Training Data (Organic + Synthetic Components).

using a validation set as described in Sec. 4.2.2.4. To reiterate, all of the images in the test and validation sets were extracted from real applications and were separate (*e.g.*, unseen) from the training set. To evaluate the effectiveness of our approach we measure the average top-1 classification precision across all classes on the Test set of data:

$$P = \frac{TP}{TP + FP}$$

where TP corresponds to true positives, or instances where the top class predicted by the network is correct, and FP corresponds to false positives, or instances where the top classification prediction of the network is not correct. To illustrate the classification capabilities of our CNN, we present a confusion matrix with precision across classes in Sec. 4.4. The confusion matrix illustrates correct true positives across the highlighted diagonal, and false positives in the other cells. To help justify the need and applicability of a CNN-based approach, we measure the classification performance of our CNN against a baseline technique, as recent work has suggested that deep learning techniques applied to SE tasks should be compared to simpler, less computationally expensive alternatives [157]. To this end, we implemented a baseline Support Vector Machine (SVM) for classification based image classification approach [149] that utilizes a "Bag of Visual Words" (BOVW). At a

high level, this approach extracts image features using the Speeded-Up Robust Feature (SURF) detection algorithm [126], then uses K-means clustering to cluster similar features together, and utilizes an SVM trained on resulting feature clusters. We utilized the same training/validation/test set of data used to train the CNN and followed the methodology in [149] to vary the number of K-means clusters from $k = 1,000$ to $k = 5,000$ in steps of 50, finding that $k = 4,250$ achieved the best performance in terms of classification precision for our dataset. We also report the confusion matrix of precision values for the BOVW technique.

4.3.2 RQ₂: GUI Hierarchy Construction

In order to answer RQ₂ we aim to measure the similarity of the GUI-hierarchies in apps generated by REDRAW compared to a ground truth set of hierarchies and a set of hierarchies generated by two baseline mobile app prototyping approaches, REMAUI and pix2code. To carry out this portion of the study, we selected 32 apps from our cleaned dataset of `Apks` by randomly selecting one of the top-10 apps from each category (grouping all "Family" categories together). We then manually extracted 2-3 screenshots and `uiautomator xml` files per app, which were not included in the original dataset used to train, validate or test the CNN. After discarding screens according to our filtering techniques, this resulted in a set of 83 screens. Each of these screens was used as input to REDRAW, REMAUI, and pix2code from which a prototype application was generated. Ideally, a comparison would compare the GUI-related source code of applications (*e.g.*, `xml` files located in the `res` folder of Android project) generated using various automated techniques however, the source code of many of the subject Google Play applications is not available. Therefore, to compare GUI-hierarchies, we compare the runtime GUI-hierarchies extracted dynamically from the generated prototype apps for each approach using `uiautomator`, to the set of "ground truth" `uiautomator xml` files extracted from the original applications. The `uiautomator` representation of the GUI is a reflection of the automatically generated GUI-related source code for each studied prototyping approach displayed at runtime on the device screen.

Table 4.2: Semi-Structured Interview Questions for Developers & Designers

Q#	Question Text
Q1	Given the scenario where you are creating a new user interface, would you consider adopting ReDraw in your design or development workflow? Please elaborate.
Q2	What do you think of the visual similarity of the ReDraw applications compared to the original applications? Please elaborate.
Q3	Do you think that the GUI-hierarchies (<i>e.g.</i> , groupings of components) generated by ReDraw are effective? Please elaborate.
Q4	What improvements to ReDraw would further aid the mobile application prototyping process at your company? Please elaborate.

This allows us to make an accurate comparison of the hierarchal representation of GUI-components and GUI-containers for each approach.

To provide a performance comparison to REDRAW, we selected the two most closely related approaches in related research literature, REMAUI [232] and pix2code [127], to provide a comparative baseline. To provide a comparison against pix2code, we utilized the code provided by the authors of the paper on GitHub [72] and the provided training dataset of synthesized applications. We were not able to train the pix2code approach on our mined dataset of Android application screenshots for two reasons: (i) pix2code uses a proprietary domain specific language (DSL) that training examples must be translated to and the authors do not provide transformation code or specifications for the DSL, (ii) the pix2code approach requires the GUI-related source code of the applications for training, which would have needed to be reverse engineered from the Android apps in our dataset from Google Play. To provide a comparison against REMAUI [232], we re-implemented the approach based on the details provided in the paper, as the tool was not available as of the time of writing this dissertation¹.

As stated in Sec. 4.2.1 REDRAW enables two different methodologies for for *detecting* GUI-components from a mock-up artifact: (i) CV-based techniques and (ii) parsing information directly from mock-up artifacts. We consider both of these variants in our evaluation which we will refer to as REDRAW-CV (for the CV-based approach) and REDRAW-Mockup (for the approach that parses mock-up metadata). Our set of 83 screens

¹REMAUI is partially available as a web-service [81], but it did not work reliably and we could not generate apps using this interface.

extracted from Google Play does not contain traditional mock-up artifacts that would arise as part of the app design and development process (*e.g.*, Photoshop or Sketch files) and reverse engineering these artifacts is an extremely time-consuming task (see Sec. [4.3.4](#)). Thus, because manually reverse-engineering mock-ups from 83 screens is not practical, REDRAW-Mockup was modified to parse *only* the bounding-box information of leaf node GUI-components from `uiautomator` files as a substitute for mock-up metadata.

We compared the runtime hierarchies of all generated apps to the original, ground truth runtime hierarchies (extracted from the original `uiautomator xml` files) by deconstructing the trees using pre-order and using the Wagner-Fischer [\[110\]](#) implementation of Levenshtein edit distance for calculating similarity between the hierarchical (*i.e.*, tree) representations of the runtime GUIs. The hierarchies were deconstructed such that the *type* and *nested order* of components are included in the hierarchy deconstruction. We implemented the pre-order traversal in this way to avoid small deviations in other attributes included in the `uiautomator` information, such as pixel values, given that the main *goal* of this evaluation is to measure hierarchical similarities.

In our measurement of edit distance, we consider three different types of traditional edit operations: insertion, deletion, and substitution. In order to more completely measure the similarity of the prototype app hierarchies to the ground truth hierarchies, we introduced a weighting schema representing a “penalty” for each type of edit operation, wherein the default case each operation carries an identical weight of 1/3. We vary the weights of each edit and calculate a distribution of edit distances which are dependent on the fraction of the total penalty that a given operation (*i.e.*, insertion, deletion, or substitution) occupies, and carry out these calculations varying each operation separately. The operations that are not under examination split the difference of the remaining weight of the total penalty equally. For example, when insertions are given a penalty of 0.5, the penalties for deletion and substitution are set to 0.25 each. This helps to better visualize the minimum edit distance required to transform a REDRAW, `pix2code`, or REMAUI generated hierarchy to the

original hierarchy and also helps to better describe the nature of the inaccuracies of the hierarchies generated by each method.

4.3.3 RQ₃: Visual Similarity

One of REDRAW’s goals is to generate apps that are visually similar to target mock-ups. Thus to answer RQ₃, we compared the visual similarity of apps generated by REDRAW, pix2code, and REMAUI, using the same set of 83 apps from RQ₂. The subjects of comparison for this section of the study were screenshots collected from the prototype applications generated by REDRAW-CV, REDRAW-Mockup, pix2code, and REMAUI. Following the experimental settings used to validate REMAUI [232], we used the open source PhotoHawk [71] library to measure the *mean squared error* (MSE) and *mean average error* (MAE) of screenshots from the generated prototype apps from each approach compared to the original app screenshots. To examine whether the MAE and MSE varied to a statistically significant degree between approaches, we compare the MAE & MSE distributions for each possible pair of approaches using a two-tailed Mann-Whitney test [145] (*p*-value). Results are declared as statistically significant at a 0.05 significance level. We also estimate the magnitude of the observed differences using the Cliff’s Delta (*d*), which allows for a nonparametric effect size measure for ordinal data [161].

4.3.4 RQ₄: Industrial Applicability

Ultimately, the goal of REDRAW is integration into real application development workflows, thus as part of our evaluation, we aim to investigate REDRAW’s applicability in such contexts. To investigate RQ₄ we conducted semi-structured interviews with a front-end Android developer at Google, an Android UI designer from Huawei, and a mobile researcher from Facebook. For each of these three participants, we randomly selected nine screens from the set of apps used in RQ₂-RQ₃ and manually reversed engineered Sketch mock-ups of these apps. We verified the visual fidelity of these mock-ups using the GVT tool [224], which has been used in prior work to detect presentation failures, ensuring that there were no reported design violations reported in the reverse-engineered mockups.

Table 4.3: Confusion Matrix for REDRAW

	Total	TV	IV	Bt	S	ET	IBt	CTV	PB	RB	TB	CB	Sp	SB	NP	RBt
TV	9877	94%	3%	2%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
IV	5345	5%	93%	1%	0%	0%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Bt	1600	11%	6%	81%	0%	1%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%
S	37	5%	3%	0%	87%	0%	0%	5%	0%	0%	0%	0%	0%	0%	0%	0%
ET	567	14%	3%	2%	0%	81%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
IBt	866	4%	23%	1%	0%	0%	72%	0%	0%	0%	0%	0%	0%	0%	0%	0%
CTV	337	7%	0%	0%	0%	0%	0%	93%	0%	0%	0%	0%	0%	0%	0%	0%
PB	41	15%	29%	0%	0%	0%	0%	0%	56%	0%	0%	0%	0%	0%	0%	0%
RB	22	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%
TBt	26	19%	22%	7%	0%	0%	0%	0%	0%	0%	52%	0%	0%	0%	0%	0%
CB	165	12%	7%	0%	0%	1%	0%	0%	0%	0%	0%	81%	0%	0%	0%	0%
Sp	2	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%
SB	39	10%	13%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	78%	0%	0%
NP	40	0%	5%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	95%	0%
RBt	129	4%	3%	2%	0%	0%	0%	1%	0%	0%	0%	1%	0%	0%	0%	89%

This process of reverse-engineering the mock-ups was extremely time-consuming to reach acceptable levels, with well over ten hours invested into each of the nine mock-ups. We then used REDRAW to generate apps using both CV-based detection and utilizing data from the mock-ups. Before the interviews, we sent participants a package containing the ReDraw generated apps, complete with screenshots and source code, and the original app screenshots and Sketch mock-ups. We then asked a series of questions (delineated in Table 4.2) related to (i) the potential applicability of the tool in their design/development workflows, (ii) aspects of the tool they appreciated, and (iii) areas for improvement. Our investigation into this research question is meant to provide insight into the applicability of REDRAW to fit into real design development workflows, however, we leave full-scale user studies and trials as future work with industrial collaborators. This study is not meant to be comparative, but rather to help gauge REDRAW’s industrial applicability.

4.4 Experimental Results

4.4.1 RQ₁ Results: Effectiveness of the CNN

The confusion matrices illustrating the classification precision across the 15 Android component classes for both the CNN-classifier and the Baseline BOVW approach are shown in Tables 4.3 & 4.4 respectively. The first column of the matrices illustrate the number of

Table 4.4: Confusion Matrix for BOVW Baseline

	Total	TV	IV	Bt	S	ET	IBt	CTV	PB	RB	TB	CB	Sp	SB	NP	RBt
TV	9877	59%	4%	9%	1%	6%	2%	8%	6%	0%	1%	2%	0%	1%	0%	2%
IV	5345	4%	51%	4%	1%	2%	11%	2%	18%	1%	1%	3%	0%	2%	0%	2%
Bt	1600	6%	6%	59%	1%	5%	4%	7%	4%	0%	1%	1%	0%	0%	3%	1%
S	37	5%	0%	3%	65%	0%	0%	5%	22%	0%	0%	0%	0%	0%	0%	0%
ET	567	6%	2%	4%	1%	62%	1%	4%	15%	0%	0%	1%	0%	0%	4%	1%
IBt	866	2%	16%	3%	0%	2%	61%	1%	9%	1%	1%	2%	0%	2%	0%	3%
CTV	337	3%	1%	7%	1%	3%	0%	81%	1%	0%	0%	2%	0%	0%	0%	2%
PB	41	0%	24%	2%	0%	2%	5%	2%	54%	0%	0%	2%	2%	2%	0%	2%
RB	22	0%	5%	0%	0%	0%	0%	0%	27%	68%	0%	0%	0%	0%	0%	0%
TBt	26	7%	7%	19%	0%	0%	0%	11%	15%	0%	33%	0%	0%	0%	0%	7%
CB	165	4%	2%	3%	1%	2%	1%	2%	12%	1%	0%	72%	0%	0%	0%	1%
Sp	2	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%
SB	39	0%	5%	0%	0%	0%	0%	0%	18%	3%	0%	5%	0%	68%	0%	3%
NP	40	3%	0%	5%	0%	3%	0%	5%	0%	0%	0%	0%	0%	0%	84%	0%
RBt	129	6%	3%	5%	1%	3%	0%	6%	18%	0%	1%	1%	0%	1%	0%	55%

Abbreviations for column headings representing GUI-component types: TextView (TV), ImageView (IV), Button (Bt), Switch (S), EditText (ET), ImageButton (IBt), Checked-TextView (CTV), ProgressBar (PB), RadioButton (RB), ToggleButton (TBt), CheckBox (CB), Spinner (Sp), SeekBar (SB), NumberPicker (NP), RadioButton (RBt)

components in the test set, and the numbers in the matrix correspond to the percentage of each class on the y-axis, that were classified as components on the x-axis. Thus, the diagonal of the matrices (highlighted in blue) corresponds to correct classifications. The overall top-1 precision for the CNN (based on raw numbers of components classified) is 91.1%, whereas for the BOVW approach the overall top-1 precision is 64.7%. Hence, it is clear that the CNN-based classifier that REDRAW employs outperforms the baseline, illustrating the advantage of the CNN architecture compared to a heuristic-based feature extraction approach. In fact, REDRAW’s CNN outperforms the baseline in classification precision *across all classes*.

It should be noted that REDRAW’s classification precision does suffer for certain classes, namely `ProgressBars` and `ToggleButtons`. We found that the classification accuracy of these component types was hindered due to multiple existing styles of the components. For instance, the `ProgressBar` had two primary styles, traditional progress bars, which are short in the y-direction and long in the x-direction, and square progress bars that rendered a progress wheel. With two very distinct shapes, it was difficult for our CNN to distinguish between the drastically different images and learn a coherent set of features to differentiate

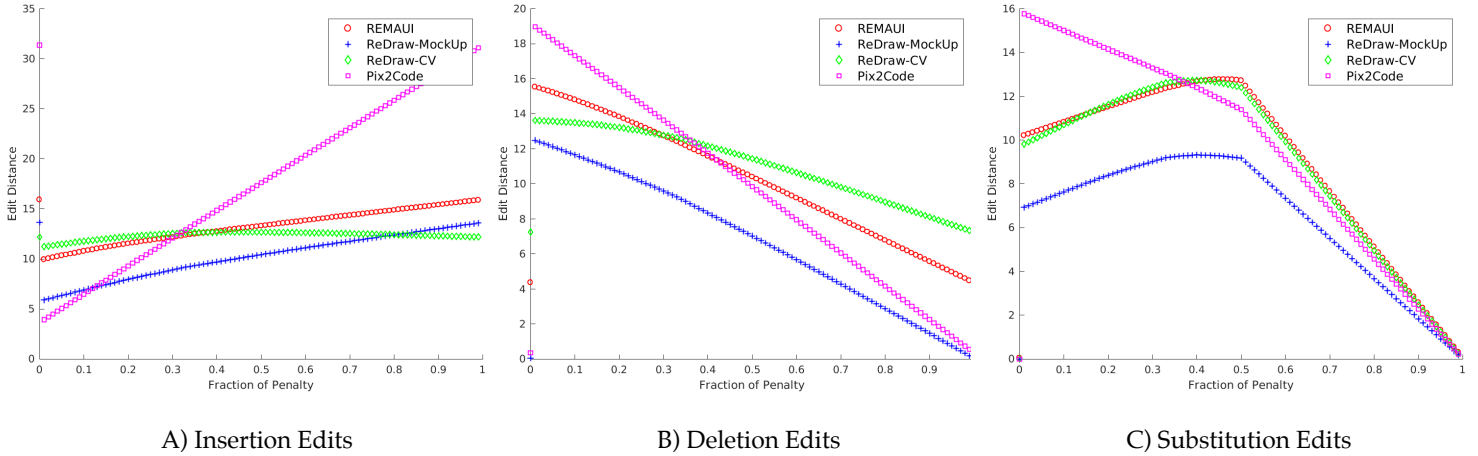


Figure 4.9: Hierarchy similarities based on edit distances

the two. While the CNN may occasionally misclassify components, the confusion matrix illustrates that these misclassifications are typically skewed toward *similar* classes. For example, `ImageButtons` are primarily misclassified as `ImageViews`, and `EditTexts` are misclassified as `TextViews`. Such misclassifications in the GUI-hierarchy would be trivial for experienced Android developers to fix in the generated app while the GUI-hierarchy and boilerplate code would be automatically generated by ReDraw. The strong performance of the CNN-based classifier provides a solid base for the application generation procedure employed by ReDraw. Based on these results, we answer **RQ₁**:

RQ₁: ReDraw’s CNN-based GUI-component classifier was able to achieve a high average precision (91%) and outperform the baseline BOVW approach’s average precision (65%).

4.4.2 RQ₂ Results: Hierarchy Construction

An important part of the app generation process is the automated construction of a GUI-hierarchy to allow for the proper grouping, and thus proper displaying, of GUI-components into GUI-containers. Our evaluation of ReDraw’s GUI-hierarchy construction compares against the REMAUI and pix2code approaches by decomposing the runtime GUI-hierarchies into trees and measuring the edit distance between the generated trees and target trees

(as described in Section 4.3.2). By varying the penalty prescribed to each edit operation, we can gain a more comprehensive understanding of the similarity of the generated GUI-hierarchies by observing, for instance, whether certain hierarchies were more or less shallow than real applications, by examining the performance of insertion and deletion edits.

The results for our comparison based on Tree edit distance are illustrated in Fig. 4.9 A-C. Each graph illustrates the results for a different edit operation and the lines delineated by differing colors and shapes represent the studied approaches (REDRAW Mock-Up or CV-based, REMAUI, or pix2code) with the edit distance (*e.g.*, closeness to the target hierarchy) shown on the y-axis and the penalty prescribed to the edit operation on the x-axis. For each of the graphs, a lower point or line indicates that a given approach was closer to the target mock-up hierarchy. The results indicate that in general, across all three variations in edit distance penalties, REDRAW-MockUp produces hierarchies that are closer to the target hierarchies than REMAUI and pix2code. Of particular note is that as the cost of insertion operations rises both REDRAW-CV and REDRAW-MockUp outperform REMAUI. In general REDRAW-Mockup requires fewer than ten edit operations across the three different types of operations to exactly match the target app’s GUI-hierarchy. While REDRAW’s hierarchies require a few edit operations to exactly match the target, this may be acceptable in practice, as there may be more than one variation of an acceptable hierarchy. Nevertheless, REDRAW-Mockup is closer than other related approaches in terms of similarity to real hierarchies.

Another observable phenomena exhibited by this data is the tendency for REMAUI and pix2code to generate relatively *shallow* hierarchies. We see that as the penalty for insertion increases, both REDRAW-CV and REDRAW-Mockup outperform REMAUI and pix2code. This is because ReDraw simply does not have to perform as many insertions into the hierarchy to match the ground truth. Pix2code and REMAUI are forced to add more inner nodes to the tree because their generated hierarchies are too shallow (*i.e.* lacking in inner nodes). From a development prototyping point of view, it is more likely easier for a developer to remove redundant nodes than it is to create new nodes, requiring them reasoning what amounts to a new hierarchy after the automated prototyping process. These results

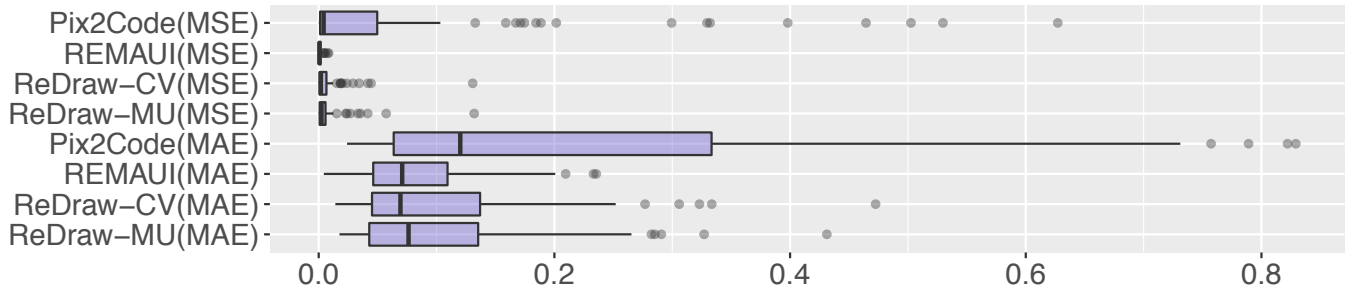


Figure 4.10: Pixel-based mean average error and mean squared error of screenshots: REDRAW, REMAUI, and pix2code apps

are unsurprising for the REMAUI approach, as the authors used shallowness as a proxy for suitable hierarchy construction. However, this evaluation illustrates that the shallow hierarchies generated by REMAUI and pix2code do match the target hierarchies as well as those generated by REDRAW-Mockup. While minimal hierarchies are desirable from the point of view of rendering content on the screen, we find that REMAUI’s hierarchies tend to be dramatically more shallow compared to REDRAW’s which exhibit higher similarity to real hierarchies. Another important observation is that the substitution graph illustrates the general advantage that the CNN-classifier affords during hierarchy construction. REDRAW-Mockup requires far fewer substitution operations to match a given target hierarchy than REMAUI, which is at least in part due to REDRAW’s ability to properly classify GUI-components, compared to the text/image binary classification afforded by REMAUI. From these results, we can answer **RQ₂**:

RQ₂: REDRAW-MockUp is capable of generating GUI-hierarchies closer in similarity to real hierarchies than REMAUI or pix2code. This signals that ReDraw’s hierarchies can be utilized by developers with low effort.

4.4.3 RQ₃ Results: Visual Similarity

An effective GUI-prototyping approach should be capable of generating apps that are visually similar to the target mock-up artifacts. We measured this by calculating the MAE and MSE across all pixels in screenshots from generated apps for ReDraw-MockUp, REDRAW-CV, REMAUI, and pix2code (Fig. 4.10.) compared to the original app screenshots.

Table 4.5: Pixel-based comparison by MAE: Mann-Whitney test (p -value) and Cliff’s Delta (d).

Test	p -value	d
ReDraw-MU vs ReDraw-CV	0.835	0.02 (Small)
ReDraw-MU vs REMAUI	0.542	0.06 (Small)
ReDraw-MU vs pix2Code	< 0.0002	-0.34 (Medium)
pix2Code vs ReDraw-CV	< 0.0001	0.35 (Medium)
pix2Code vs REMAUI	< 0.0001	0.39 (Medium)
REMAUI vs ReDraw-CV	0.687	-0.04 (Small)

Table 4.6: Pixel-based comparison by MSE: Mann-Whitney test (p -value) and Cliff’s Delta (d).

Test	p -value	d
ReDraw-MU vs ReDraw-CV	0.771	0.03 (Small)
ReDraw-MU vs REMAUI	< 0.0001	0.45 (Medium)
ReDraw-MU vs pix2Code	< 0.003	-0.27 (Small)
pix2Code vs ReDraw-CV	< 0.002	0.28 (Small)
pix2Code vs REMAUI	< 0.0001	0.61 (Large)
REMAUI vs ReDraw-CV	<0.0001	-0.42 (Medium)

This figure depicts a box-and-whisker plot with points corresponding to a measurement for each of the studied 83 subject applications. The black bars indicate mean values. In general, the results indicate that all approaches generated apps that exhibited high overall pixel-based similarity to the target screenshots. REDRAW-CV outperformed both REMAUI and pix2code in MAE, whereas all approaches exhibited very low MSE, with REMAUI very slightly outperforming both ReDraw variants. The apps generated by pix2code exhibit a rather large variation from the target screenshots used as input. This is mainly due to the artificial nature of the training set utilized by pix2code which in turn generates apps only with a relatively rigid, pre-defined set of components. The results of the Mann-Whitney test reported in Table 4.5 & 4.6 illustrate whether the similarity between each combination of approaches was statistically significant. For MAE, we see that when REDRAW-CV and REDRAW-Mockup are compared to REMAUI, the results are not statistically significant, however, when examining the MSE for these same approaches the result is statistically significant with a medium effect size according to the Cliff’s delta measurement. Thus, it is clear that on average REDRAW and REMAUI both generate prototype applications

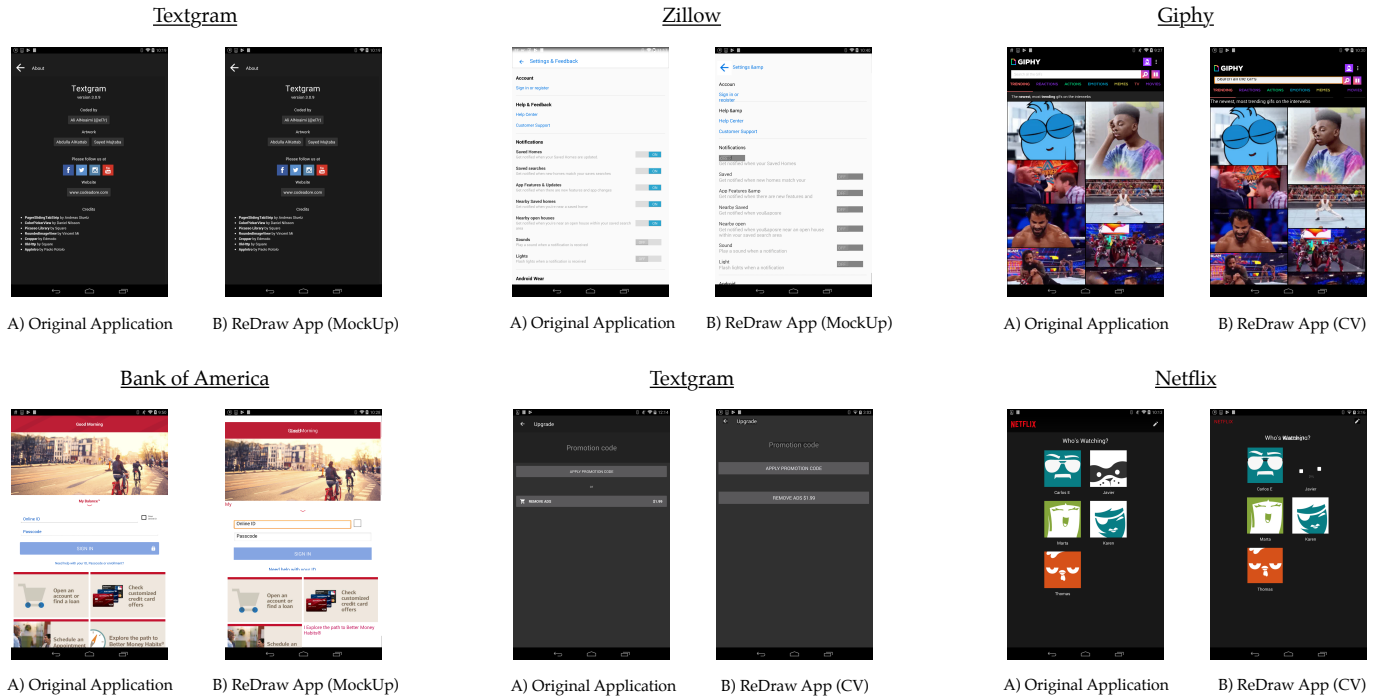


Figure 4.11: Examples of apps generated with REDRAW exhibiting high visual and structural similarity to target apps

that are closely similar to a target visually, with REMAUI outperforming REDRAW in terms of MSE to a statistically significant degree (with the overall MSE being extremely low < 0.007 for both approaches) and REDRAW outperforming REMAUI in terms of average MAE (although not to a statistically significant degree). This is encouraging, given that REMAUI directly copies images of components (including those that are not images, like buttons) and generates text-fields. Reusing images for all non-text components is likely to lead to more visually similar (but less functionally accurate) apps than classifying the proper component type and inferring the stylistic details of such components. When comparing both variants of REDRAW and REMAUI to pix2code, the results are all statistically significant, with ranging effect sizes. Thus, both REDRAW and REMAUI outperform pix2code in terms of generating prototypes that are visually similar to a target.

While in general the visual similarity for apps generated by REDRAW is high, there are instances where REMAUI outperformed our approach. Typically this is due to instances

where REDRAW misclassifies a small number of components that cause visual differences. For example, a button may be classified and rendered as a switch in rare cases. However, REMAUI does not suffer from this issue as all components deemed not to be text are copied to the generated app as an image. While this occasionally leads to more visually similar apps, the utility is dubious at best, as developers will be required to add proper component types, making extensive edits to the GUI-code. Another instance that caused some visual inconsistencies for REDRAW was text overlaid on top of images. In many cases, a developer might overlay a snippet of text over an image to create a striking effect (*e.g.*, Netflix often overlays text across movie-related images). However, this can cause an issue for REDRAW’s prototyping methodology. During the detection process, REDRAW recognizes images and overlaid text in a mockup. However, given the constraints of our evaluation, REDRAW simply re-uses the images contained within screenshot as is, which might include overlaid text. Then, ReDraw would render a `TextView` or `EditText` over the image *which already includes the overlaid text* causing duplicate lines of text to be displayed. In a real-world prototyping scenario, such issues can be mitigated by designers providing “clean” versions of the images used in a mockup, so that they could be utilized in place of “runtime” images that may have overlaid text. Overall, the performance of REDRAW is quite promising in terms of the visual fidelity of the prototype apps generated, with the potential for improvement if adopted into real design workflows.

We illustrate some of the more successful generated apps (in terms of visual similarity to a target screenshot) in Fig. 4.11; screenshots and hierarchies for all generated apps are available in a dataset in our online appendix [80]. In summary, we can answer **RQ₃** as follows:

RQ₃: The apps generated by ReDraw exhibit high visual similarity compared to target screenshots.
--

4.4.4 RQ₄ Results: Industrial Applicability

To understand the applicability of REDRAW from an industrial perspective we conducted a set of semi-structured interviews with a front-end Android developer @Google, a mobile designer @Huawei, and a mobile researcher @Facebook. We asked them four questions (see Sec. 4.3) related to (i) the applicability of REDRAW, (ii) aspects of REDRAW they found beneficial, and (iii) areas for improvement.

4.4.4.1 Front End Android Developer @Google

The first individual works mostly on Google’s search products, and his team practices the process of mock-up driven development, where developers work in tandem with a dedicated UI/UX team. Overall, the developer was quite positive about REDRAW explaining that it could help to improve the process of writing a new Android app activity from scratch, however, he noted that *“It’s a good starting point... From a development standpoint, the thing I would appreciate most is getting a lot of the boilerplate code done [automatically]”*. In the “boilerplate” code statement, the developer was referring to the large amount of layout and style code that must be written when creating a new activity or view. He also admitted that this code is typically written by hand stating, *“I write all my GUI-code in xml, I don’t use the Android Studio editor, very few people use it”*. He also explained that this GUI-code is time-consuming to write and debug stating, *“If you are trying to create a new activity with all its components, this can take hours”*, in addition to the time required for the UI/UX team to verify proper implementation. The developer did state that some GUI-hierarchies he examined tended to have redundant containers, but that these can be easily fixed stating, *“There are going to be edge cases for different layouts, but these are easily fixed after the fact”*.

The aspect of REDRAW that this developer saw the greatest potential for, is its use in an evolutionary context. During the development cycle at Google, the UI/UX team will often propose changes to existing apps, whose GUI-code must be updated accordingly. The developer stated that REDRAW had the potential to aid this process: *“The key thing is fast*

iteration. A developer could generate the initial view [using ReDraw], clean up the layouts, and have a working app. If a designer could upload a screenshot, and without any other intervention [ReDraw] could update the [existing] xml this would be ideal." The developer thought that if REDRAW was able to detect existing GUI-components in a prior app version, and update the layouts and styles of these components according to a screenshot, generating new components as necessary, this could greatly improve the turn around time of GUI-changes and potentially increase quality. He even expressed optimism that the approach could learn from developer corrections on generated code over time, stating *"It would be great if you could give it [ReDraw] developer fixes to the automatically generated xml and it could learn from this."*

4.4.4.2 Mobile UI/UX Designer @Huawei

We also interviewed a dedicated UI/UX designer at Huawei, with limited programming experience. His primary job is to create mock-up artifacts that stipulate designs of mobile apps, communicate these to developers, and ensure they are implemented to spec. This interview was translated from Chinese into English. This designer also expressed interest in REDRAW, stating that the visual similarity of the apps was impressive for an automated approach, *"Regarding visual, I feel that it's very similar"*, and that such a solution would be sought after at Huawei, *"If it [a target app] can be automatically implemented after the design, it should be the best design tool [we have]"*. While this designer does not have extensive development experience, he works closely with developers and stated that the quality of the reusability of the code is a key point for adoption, *"In my opinion, for the developers it would be ideal if the output code can be reused"*. This is promising as REDRAW was shown to generate GUI-hierarchies that are comparatively more similar to real apps than other approaches.

4.4.4.3 Mobile Researcher @Facebook

The last participant was a mobile systems researcher at Facebook. This participant admitted that Facebook would most likely not use REDRAW in its current state, as they are

heavily invested in the React Native ecosystem. However, he saw the potential of the approach if it were adopted for this domain, stating “*I can see this as a possible tool to prototype designs*”. He was impressed by the visual similarity of the apps, stating, “*The visual similarity seems impressive*”.

In the end, we can answer **RQ₄**:

RQ₄: REDRAW has promise for application into industrial design and development workflows, particularly in an evolutionary context. However, modifications would most likely have to be made to fit specific workflows and prototyping toolchains.

4.5 Limitations & Threats to Validity

In this section we describe some limitations and possible routes for future research in automated software prototyping, along with potential threats to validity of our approach and study.

4.5.1 Limitations and Avenues for Future Work

While REDRAW is a powerful approach for prototyping GUIs of mobile apps, it is tied to certain practical limitations, some of which represent promising avenues for future work in automated software prototyping. First, REDRAW is currently capable of prototyping a single screen for an application, thus if multiple screens for a single app are desired, they must be prototyped individually and then manually combined into a single application. It would be relatively trivial to modify the approach and allow for multiple screens within a single application with a simple swipe gesture to switch between them for software demo purposes however, we leave this a future work. Additionally, future work might examine a learning-based approach for prototyping and linking together multiple screens, learning common app transitions via dynamic analysis and applying the learned patterns during prototyping.

Second, the current implementation of KNN-hierarchy construction is tied to the specific screen size of the devices used during the data-mining and automated dynamic analysis. However, it is possible to utilize display independent pixel (dp) values to generalize this algorithm to function independently of screen size, we leave this as future work.

Third, as discussed in Section [4.2.3.2](#), REDRAW is currently limited to detecting and assembling a distinct set of stylistic details from mock-up artifacts including: (i) background colors; (ii) font colors, and (iii) font sizes. REDRAW was able to produce prototype applications that exhibited high visual similarity to target screenshots using only these inferences. However, a promising area for future work on automated prototyping of software GUIs involves expanding the stylistic details that can be inferred from a target mock-up artifact. Future work could perform more detailed studies on the visual properties of individual components from prototype screens generated from screenshots of open source apps. This study could then measure how well additional inferred styles of individual components match the original developer implemented components.

Our current CNN classifier is capable of classifying incoming images into one of 15 of the most popular Android GUI-components. Thus, we do not currently support certain, rarely used component types. Future work could investigate network architectures with more capacity (*e.g.*, deeper architectures) to classify larger numbers of component types, or even investigate emerging architectures such as Hierarchical CNNs [\[263\]](#). Currently, REDRAW requires two steps for *detecting* and *classifying* components, however, future approaches could examine the applicability of CNN-based object detection networks [\[240\]](#), [\[158\]](#) that may be capable of performing these two steps in tandem.

4.5.2 Internal Validity

Threats to *internal validity* correspond to unexpected factors in the experiments that may contribute to observed results. One such threat stems from our semi-structured interview with industrial developers. While evaluating industrial applicability of REDRAW, threats may arise from our manual reverse engineering of Sketch mock-ups. However, we applied

a state of art tool for detecting design violations in GUIs [224] in order to ensure their validity, sufficiently mitigating this threat.

During our experimental investigation of RQ₂-RQ₄, we utilized two different types of mock-up artifacts, (i) images of existing application screens (RQ₂ & RQ₃, and (ii) reverse engineered mock-ups from existing application screens. The utilization of these artifacts represents a threat to internal validity as they are used as a proxy for real mock-up artifacts. However, real mock-ups created during the software design process may exhibit some unique characteristics not captured by these experimental proxies. For example, software GUI designs can be highly fluid, and oftentimes, may not be complete when handed off to a developer for implementation. Furthermore, real mock-ups may stipulate a design that cannot be properly instantiated in code (*i.e.*, unavailable font types, components organized in spatial layouts that are not supported in code). We acknowledge that our experiments do not measure the performance of REDRAW in such cases. However, collecting real mock-up artifacts was not possible in the scope of our evaluation, as they are typically not included in the software repositories of open source applications. We performed a search for such artifacts on all Android projects hosted on GitHub as of Spring 2017, and found that no repository contained mock-ups created using Sketch. As stated earlier, it was not practically feasible to reverse-engineer mock-ups for all 83 applications utilized in our dataset for these experiments. Furthermore, these screenshots represent production-grade app designs that are used daily by millions of users, thus we assert that these screenshots and mock-ups represent a reasonable evaluation set for REDRAW. We also did not observe any confounding results when applying REDRAW to our nine reverse engineered Sketch mock-ups, thus we assert that this threat to validity is reasonably mitigated.

Another potential confounding factor is our dataset of labeled components used to train, validate, and test the CNN. To help ensure a correct, coherent dataset, we applied several different data filtering, cleaning, and augmentation techniques, inspired by past work on image classification using CNNs described in detail in Sec. 4.2.2.4. Furthermore, we utilized the `uiautomator` tool included in the Android SDK, which is responsible for

reporting information about runtime GUI-objects, and is generally accurate as it is tied directly to Android sub-systems responsible for rendering the GUI. To further ensure the validity of our dataset, we randomly sampled a statistically significant portion of our dataset and manually inspected the labeled images *after* our data-cleaning process was applied. We observed no irregularities and thus mitigating a threat related to the quality of the dataset. It is possible that certain components can be specifically styled by developers to look like other components (*e.g.*, a textview styled to look like a button) that could impact the CNN component classifications. However, our experiments illustrate that in our real-world dataset overall accuracy is still high, suggesting that such instances are rare. Our full dataset and code for training the CNN are available on REDRAW’s website to promote reproducibility and transparency [80].

During our evaluation of REDRAW’s ability to generate suitable GUI-hierarchies, we compared them against the actual hierarchies of the original target applications. However, it should be noted, that the notion of a correct hierarchy may vary between developers, as currently, there is no work that empirically quantifies what constitutes a *good* GUI-hierarchy for Android applications. For instance, some developers may prefer a more rigid layout with fewer container components, whereas others may prefer more components to ensure that their layout is highly reactive across devices. We compared the hierarchies generated by ReDraw to the original apps to provide an objective measurement on actual implementations of popular apps, which we assert provides a reasonable measurement of the effectiveness of REDRAW’s hierarchy construction algorithm. It should also be noted that performing this comparison on apps of different popularity levels may yield different results. We chose to randomly sample the apps from the top-10 of each Google Play category, to investigate whether REDRAW is capable of assembling GUI-hierarchies of “high-quality” apps as measured by popularity.

4.5.3 Construct Validity

Threats to *construct validity* concern the operationalization of experimental artifacts. One potential threat to construct validity lies in our reimplementation of the REMAUI tool. As stated earlier, the original version of REMAUI’s web tool was not working at the time of writing this dissertation. We reimplemented REMAUI according to the original description in the paper, however we excluded the list generation feature, as we could not reliably re-create this feature based on the provided description. While our version may vary slightly from the original, it still represents an unsupervised CV-based technique against which we can compare REDRAW. Furthermore, we offer our reimplementation of REMAUI (a Java program with opencv [69] bindings) as an open source project [80] to facilitate reproducibility and transparency in our experimentation.

Another potential threat to construct validity lies in our operationalization of the pix2code project. We closely followed the instructions given in the README of the pix2code project on GitHub to train the machine translation model and generate prototype applications. Unfortunately, the dataset used to train this model differs from the large scale dataset used to train the REDRAW CNN and inform the KNN-hierarchy construction, however, this is due to the fact pix2code requires the source code of training applications and employs a custom domain specific language, leading to incompatibilities to our dataset. We include the pix2code approach as a comparative baseline in our empirical investigation as it is one of the few approaches aimed at utilizing ML to perform automated GUI prototyping, and utilizes an architecture based purely upon neural machine translation, differing from our architecture. However, it should be noted that if trained on a proper dataset, with more advanced application assembly techniques, future work on applying machine translation to automated GUI-prototyping may present better results than those reported in this paper for pix2code.

4.5.4 External Validity

Threats to *external validity* concern the generalization of the results. While we implemented REDRAW for Android and did not measure its generalization to other domains, we believe the general architecture that we introduce with REDRAW could transfer to other platforms or types of applications. This is tied to the fact that other GUI-frameworks are typically comprised sets of varying types of widgets, and GUI-related information can be automatically extracted via dynamic analysis using one of a variety of techniques including accessibility services [160]. While there are likely challenges that will arise in other domains, such as a higher number of component types and the potential for an imbalanced dataset, we encourage future work on extending ReDraw to additional domains.

REDRAW relies upon automated dynamic analysis and scraping of GUI-metadata from explored application screens to gather training data for its CNN-based classifier. However, it is possible that other application domains do not adequately expose such metadata in an easily accessible manner. Thus, additional engineering work or modification of platforms may be required in order to effectively extract such information. If information for a particular platform is difficult to extract, future work could look toward transfer learning as a potential solution. In other words, the weights for a network trained on GUI metadata that is easily accessible (*e.g.*, from Android apps) could then be fine-tuned on a smaller number of examples from another application domain, potentially providing effective results.

4.6 Conclusion & Future Work

In this chapter we have presented a data-driven approach for automatically prototyping software GUIs, and an implementation of this approach in a tool called REDRAW for Android. A comprehensive evaluation of REDRAW demonstrates that it is capable of (i) accurately detecting and classifying GUI-components in a mock-up artifact, (ii) generating hierarchies that are similar to those that a developer would create, (iii) generating apps that are visually similar to mock-up artifacts, and (iv) positively impacting industrial

workflows. In the future, we are planning on exploring CNN architectures aimed at object detection to better support the detection task. Additionally, we are planning on working with industrial partners to integrate REDRAW, and our broader prototyping approach, into their workflows.

Chapter 5

Automatically Detecting, Reporting, and Reproducing Android Application Crashes

Continued growth in the mobile hardware and application marketplace is being driven by a landscape where users tend to prefer mobile smart devices and apps for tasks over their desktop counterparts. The gesture-driven nature of mobile apps has given rise to new challenges encountered by programmers during development and maintenance, specifically with regard to testing and debugging [180]. One of the most difficult [128, 131] and important maintenance tasks is the creation and resolution of bug reports [162]. Reports concerning app crashes are of particular importance to developers, because crashes represent a jarring software fault that is directly user facing and immediately impacts an app's utility and success. If an app is not behaving as expected due to crashes, missing features, or other bugs, nearly half of users are likely to abandon the app for a competitor [63] in marketplaces like GooglePlay [50].

Mobile developers heavily rely on user reviews [183, 237, 198], crash reports from the field in the form of stack traces, or reports in open source issue tracking systems to detect bugs in their apps. In each of these cases, the bug/crash reports are typically

lacking in information [140, 180], containing only a stack trace, overly detailed logs or loosely structured natural language (NL) information regarding the crash [129]. This is not surprising as previous studies showed that information, which is most useful for a developer resolving a bug report (*e.g.*, reproduction steps, stack traces and test cases), is often the most difficult information for reporters to provide [181]. Furthermore, the absence of this information is a major cause of developers failing to reproduce bug/crash reports [128]. In addition to the quality of the reports, some other factors specific to Android apps such as hardware and software fragmentation [7], API instability and fault-proneness [200, 125], the event-driven nature of Android apps, gesture-based interaction, sensor interfaces, and the possibility of multiple contextual states (*e.g.*, wifi/GPS on/off) make the process of detecting, reporting, and reproducing crashes challenging.

Motivated by these current issues developers face regarding mobile application crashes, we designed and implemented CRASHSCOPE, a practical system that automatically discovers, reports, and reproduces crashes for Android applications. CrashScope explores a given app using a systematic input generation algorithm and produces expressive crash reports with explicit steps for reproduction in an easily readable natural language format. This approach requires only an `.apk` file and an Android emulator or device to operate and requires no instrumentation of the subject apps or the Android OS. The entirety of the CrashScope workflow is completely automated, requiring no developer intervention, other than reading produced reports. Our systematic execution includes different exploration strategies, aimed at eliciting crashes from Android apps, which include automatic text generation capabilities based on the context of allowable characters for text entry fields, and targeted testing of contextual features, such as the orientation of the device, wireless interfaces, and sensors. We specifically tailored these features to test the common causes of app crashes as identified by previous studies [273, 194, 136]. During execution, CrashScope captures detailed information about the subject app, such as the inputs sent to the device, screenshots and GUI component information, exceptions, and crash informa-

tion. This information is then translated into detailed crash reports and replayable scripts, for any encountered crash.

This chapter makes the following noteworthy contributions:

1. We design and implement a practical and automatic approach for discovering, reporting, and reproducing Android application crashes, called CRASHSCOPE. To the best of the author’s knowledge, this is the first approach that is able to generate expressive, detailed crash reports for mobile apps, including screenshots and augmented NL reproduction steps, in a completely automatic fashion. CrashScope is also one of the only available fully-automated Android testing approaches that is practical from a developers’ perspective, requiring no instrumentation of the subject apps or OS. Our approach builds upon prior research in automated input generation for mobile apps, and implements several exploration strategies, informed by lightweight static analysis that are able to effectively detect crashes and exceptions;
2. We perform a detailed evaluation of the crash detection abilities of CrashScope on 61 Android apps as compared to five state-of-the-art Android input generation tools (Dynodroid [208], Gui-Ripper [120], PUMA [166], A³E [124], and Android Monkey [13]). Our results show that CrashScope performs at least as well as current tools in terms of detecting crashes, while automatically generating detailed reports and replayable scripts;
3. We design and carry out a user study evaluating the *reproducibility* and *readability* of our automatically generated bug reports through comparison to human written crash reports for eight open source apps. The results indicate that CrashScope reports offer more detail, while being at least as useful as the human written reports;
4. We make our experimental data, crash reports, and demo videos available in our online appendix [35].

5.1 Background & Motivation

In this section, we discuss the findings of previous studies examining mobile app bugs and crashes and then outline the limitations of the automated input generation approaches described in Chapter 2 while illustrating CrashScope’s novelty in context. Several approaches for detecting and reproducing crashes are available in literature [147, 235, 148, 236, 205, 150, 266, 279, 277, 276, 272, 251, 204, 184, 177, 176]; however, we forgo discussion of these approaches, as they are not presented in the context of mobile apps, and hence do not consider the unique associated challenges.

5.1.1 Previous Studies on Mobile App Bug/Crashes

Motivating factors from Mobile App Bug/Crash Studies aided us in designing CRASHSCOPE. Two studies stand out in terms of providing information to drive design decisions for our approach. First, Ravindranath *et al.* [239] conducted a study of 25 million real-world crash reports collected from Windows Phone users “in the wild” by the “Windows Phone Error Reporting System” (WPER). Although this study was conducted regarding crashes from a different mobile OS, several of the findings reported in this study are relevant in the context of Android, due to platform similarities: 1) a small number of root causes cover a majority of the crashes examined; 2) many crashes can be mapped to well-defined externally inducible faults, for example, HTTP errors caused by network connectivity issues; 3) the dominant root causes can affect many different user execution paths in an app. The most salient piece of information that can be gleaned from the study and applied in the design of CrashScope is the following: *An effective crash discovery tool must be able to test different contextual states in a targeted manner, while remaining resilient to encountered crashes so as to uncover crashes present in different program event-sequence paths.* We explain how CrashScope achieves targeted testing of contextual states using program analysis in Sec. 5.2

In addition, Zaeem *et al.* [273] conducted a bug study on 106 bugs drawn from 13 open-source Android applications, with the goal of identifying opportunities for automatically generating test cases that include oracles. Most notably, the results of this study were formulated as a categorization of different Android app bugs. Specifically, these categorizations were grouped into three headings: Basic Oracles, App-Agnostic Oracles, and App-Specific Oracles. CrashScope uses the well-defined oracles of uncaught exceptions and app crashes to detect faults; however, some of the bug categorizations in this study are useful in triggering these, specifically the app-agnostic categorizations of *Rotation*, *Activity Life-Cycle*, and *Gestures*. Specifically, we implemented a targeted (*i.e.*, localized) version of the double-rotation feature [273].

5.1.2 Limitations of Mobile Testing Approaches

While significant progress has been made in the area of testing and automatically generating inputs for mobile applications, the available tools generally exhibit some noteworthy limitations that inspired the development of CRASHSCOPE:

- Previous approaches lack the ability to provide detailed, easy-to-understand testing results for faults discovered during automatic input generation, leaving the developer to sort through and comprehend stack traces, log files, and non-expressive event sequences [142];
- Most approaches for automated input generation are not practical for developers to use, typically due to instrumentation or difficult setup procedures. This is affirmed by the fact developers typically prefer manual over automated testing approaches [185, 180]. As we show, instrumentation can contribute to a higher than necessary developer effort in parsing results from automated approaches.;
- Few approaches combine different strategies and features for testing apps through supporting different strategies for user text input and contextual states (*e.g.*, wifi on/off) in a single holistic approach.

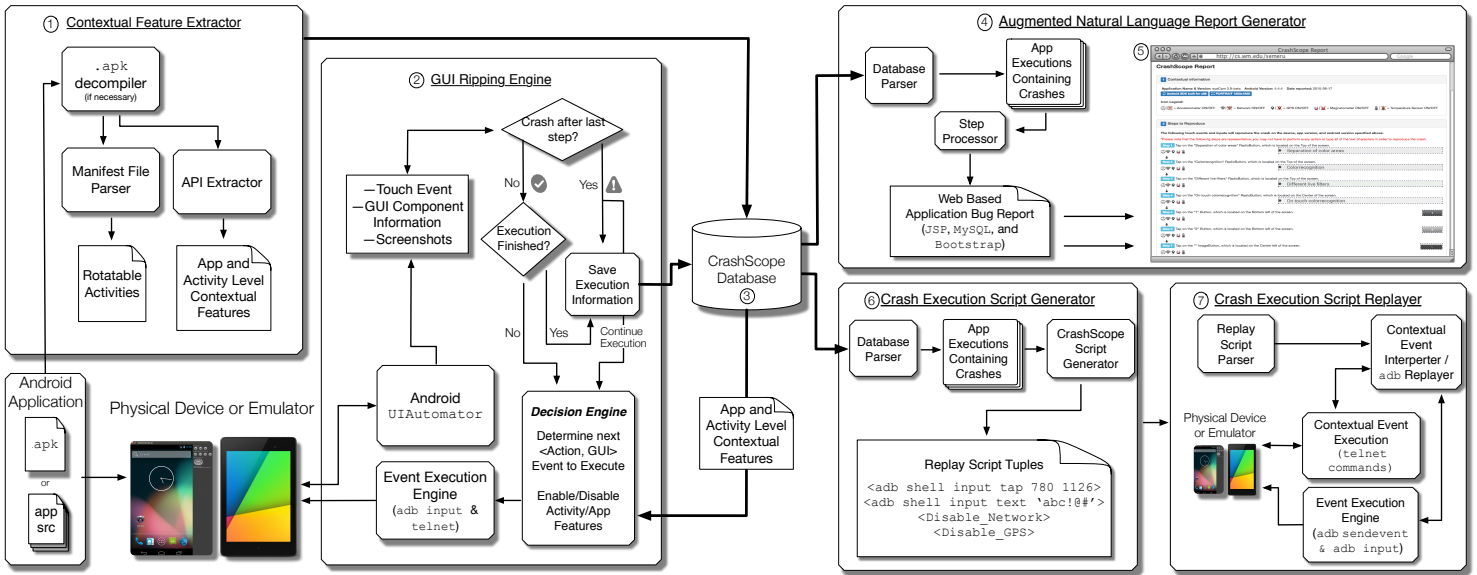


Figure 5.1: Overview of CrashScope Workflow

These shortcomings contribute to the low adoption rate of automated testing approaches by mobile developers. In the next section of this chapter, we clearly describe how CrashScope’s design addresses these current limitations in automated mobile input generation and testing tools.

5.2 CrashScope Design

In this section, we first describe CrashScope’s novelty by illustrating how it addresses the limitations discussed in the previous section. We then give an overview of the CrashScope’s workflow, and the salient features in detail.

CrashScope addresses the general limitations of existing tools. First, no other automated testing approach, is able to automatically generate *expressive* bug reports (and replayable scripts) for exceptions and crashes discovered by automated input generation for mobile apps. CrashScope accurately detects crashes and is able to generate *easily readable and detailed* reports without any developer intervention. Second, CrashScope is a practical tool, requiring only an `.apk` file and an instantiated emulator *or* physical device running Android 4.3 and newer, which constitutes 55% of the current Android OS install base[10].

Operating on emulators CrashScope is able to parallelize testing on multiple emulators with different specifications, versions of Android, and screen sizes, mitigating a major challenge in app development [180]. Third, inspired by existing approaches [239, 273, 119] CrashScope is able to explore an app through automated input generation while testing varying contextual states. We extend previous context aware testing techniques by leveraging static analysis to extract targeted locations for testing apps in different contextual states. Finally, our approach is *app-crash-resilient*; it can detect a crash and continue testing the unvisited components and states of the GUI after handling the crash.

The overall workflow of CrashScope is illustrated in Figure 5.1. Let us consider the 31C3 Schedule app [2] as a running example to explain the CRASHSCOPE workflow; then, we will discuss the salient features in detail. The first step in running CrashScope is to obtain the source code of the app, either directly or through decompilation, and detect Activities (by means of static analysis) that are related to contextual features (Figure 5.1-①) in order to target the testing of such features. In other words, CrashScope will only test certain contextual app features (*e.g.*, wifi off) if it finds instances where they are implemented in the source code. In the case of 31C3 Schedule, the first activity (screen) of the app makes use of network connectivity, so this screen would be marked as implementing this feature. More details about the contextual features detection are provided in Sec. 5.2.1.

Next, the *GUI Ripping Engine* (Figure 5.1-②) systematically executes the app using various strategies (Section 5.2.4), including enabling and disabling the contextual features (if run on an emulator) at the Activities of the app identified previously. If during the execution, uncaught exceptions are thrown, or the app crashes, dynamic execution information is saved to the CRASHSCOPE's database (Figure 5.1-③), including detailed information regarding each event performed during the systematic exploration. In the case of 31C3 Schedule, if systematic execution is continued from the first screen when the network is disabled, a crash occurs. This is because the differing contextual condition exposes a state of the app that would not be otherwise seen.

After the execution data has been saved to the CrashScope database, the *Natural Language Report Generator* (Figure 5.1-4, Section 5.2.5) parses the database and processes the information for each step of all executions that ended in a crash, generating an HTML based natural language crash report with expressive steps for reproduction (Figure 5.1-5). In addition, the *Crash Script Generator* (Figure 5.1-6, Section 5.2.6) parses the database and extracts the relevant information for each step in a crashing execution in order to create a replayable script containing `adb input` commands and markers for contextual state changes. The *Script Replayer* (Figure 5.1-7, Section 5.2.6) is able to replay these scripts by executing the sequence of `adb input` commands and interpreting the contextual state change signals, in order to reproduce the crash. In the case of the 31C3 Schedule app, this involves turning off the network connection and trying to interact with one of the app menu headers.

5.2.1 Extracting Activity and App-Level Contextual Features

CrashScope uses Abstract Syntax Tree (AST) based analysis to extract the API-call chains that are involved in invocations of contextual features. In particular, it detects Android API calls related to network connectivity and sensors (*i.e.*, Accelerometer, Magnetometer, Temperature Sensor, and GPS). Because the API calls might not be executed directly by an Activity, CrashScope performs a call-graph analysis to extract paths ending in a method invoking a contextual API. Because certain API calls may not be traceable through a back-propagated call-chain (*e.g.*, sensor or network implemented as a service), CrashScope employs two granularities for testing contextual features: activity (screen-) level and app-level. If a particular API call related to one of the contextual features above is able to be traced back to an Activity, then that feature is later tested at the Activity level (*i.e.*, the contextual feature is enabled or disabled when the corresponding Activity is in foreground). If the feature is not able to be linked to an Activity, then the feature is tested at the level of the entire app (*i.e.*, the contextual feature is enabled or disabled at the beginning of the app's execution). To obtain the activities that are rotatable, CrashScope parses the

AndroidManifest.xml, where rotatable activities must be declared. During testing, if a rotatable activity is encountered while exploring an app, then the screen is rotated from portrait to landscape and back again before any GUI interactions occur to test for proper implementation of the corresponding rotation event-handlers.

5.2.2 Exploration of Apps & Crash Detection

To explore an app, CrashScope dynamically extracts the GUI hierarchy of each app screen visited during the exploration and identifies the clickable and long-clickable components to execute, as well as available components for text inputs (*e.g.*, EditText boxes). The (long-) clickable components are added to a working list to assure that all the clickable components are executed systematically. CrashScope executes each possible event (*i.e.*, action on an available GUI component) on the current screen according to the GUI hierarchy. If text entry fields are available in a particular app screen, then each text box is filled in before each (long-) clickable component on the screen is exercised. Currently, our *Ripping Engine* supports the *tap*, *long-tap*, and *type* events.

Text entry from the user is a major part of functionality in many Android apps, therefore, CRASHSCOPE's *GUI Ripping Engine* employs a unique text input generation mechanism. CrashScope detects the type of text expected (*e.g.*, numbers) by a text field, by querying the keyboard type associated to the text field [8]. This is done with the `adb shell dumpsys input_method` command. Once the type of expected input is detected, CrashScope employs two strategies to generate text inputs: *expected* and *unexpected*. The *expected* strategy generates a string within the keyboard parameters without any punctuation or special characters, whereas the *unexpected* strategy generates random strings with all of the allowable special characters for a given keyboard type. The intuition behind this input generation mechanism is to test instances where a developer may have unknowingly set a keyboard that allows certain characters, but does not properly check for these characters in the code, resulting in a fault. Before the keyboard metadata is read, a *touch* event is executed on the text box to ensure the corresponding keyboard is displayed.

In addition to the text input generation strategies, CrashScope traverses the GUI hierarchy either from the bottom of the hierarchy up or from the top of the hierarchy down. The rationale for having two such strategies is to generally mimic what a user would do, *i.e.*, executing GUI events without a predefined order. If a transition to another screen is recorded during the exploration, then the GUI-hierarchy of the new screen is detected and the components on the new screen are executed next. The *GUI Ripping Engine* constructs a graph containing all of the possible transition states and uses the back button to return to previous states after the executable components in a particular branch have been exhausted. It also keeps a stack of all the yet-to-be visited components. To detect and capture exceptions, CrashScope filters the logcat for uncaught exceptions related only to the app being tested. To detect crashes, CrashScope checks for the appearance of the standard Android crash dialog. If a crash is encountered, the execution information is logged to the database, but because of the transition diagram and stack of unvisited components, execution can continue towards additional remaining program paths without starting the execution from scratch.

5.2.3 Testing Apps in Different Contextual States

When the GUI-Ripping begins, CrashScope first checks for app-level contextual features that should be tested according to the exploration strategy. Then, the *GUI Ripping Engine* checks if the current Activity is suitable for exercising a particular contextual feature in adverse conditions. If this is the case, it sets the value of the sensor according to the current strategy. The testing of contextual features works *only on* emulators using telnet commands associated with standard Android Virtual Devices (AVDs) [6]. While the telnet commands do support turning on/off the network for an emulator, they do not support the enabling/disabling of sensors (Accelerometer, Magnetometer, GPS, Temperature Sensor), but it is possible to set the values of these sensors. Therefore, to test for sensor related features in adverse conditions, the network connection is disabled, and unexpected values are set for the other sensors (GPS, Accelerometer, etc) that would not typically be possible

under normal conditions. For instance, to test the GPS in an adverse contextual state, CrashScope sets the value to coordinates that do not represent physical GPS coordinates.

5.2.4 Multiple Execution Strategies

One of CrashScope’s most powerful features is its ability to explore an app according to several different strategies through combinations of its various supported testing features. These strategies stem from three major feature heuristics: 1) the direction in which to traverse the GUI Hierarchy (top-down or bottom-up), 2) the method by which inputs are generated for user text entry fields (*no text*, *expected text*, *unexpected text*), and finally, 3) enabling or disabling the testing of adverse contextual states (*e.g.*, if an activity is found to have utilize wifi, should it be turned on or off?). Different combinations of these strategies have the potential to uncover different types of app crashes. For example, consider the following configuration `<no_text, top_down, enable_all_context_states>`. According to this strategy, CrashScope will not enter any user text, will exercise the GUI-components in order from the top of the screen to the bottom, and will trigger adverse contextual features in activities where they are detected. This type of strategy has a high likelihood of uncovering crashes like the one described earlier in C13C Schedule in which the change of contextual state triggers a crash. However, the `<unexpected_text, top_down, disable_context_states>` has a better chance of uncovering crashes related to user input being handled improperly by the app. By running an app through all 12 combinations of these three feature heuristics in different strategies, CrashScope can effectively test for different types of commonly inducible crashes. These strategies can also be parallelized by running several strategies for an app concurrently on a group or cloud of emulator instances, further reducing the testing overhead for the developer.

5.2.5 Generating Expressive, Natural Language Crash Reports

CrashScope generates a Crash Report (Figure [5.1-5](#)) that contains four major types of information: 1) general information including the app name and version, the version of the

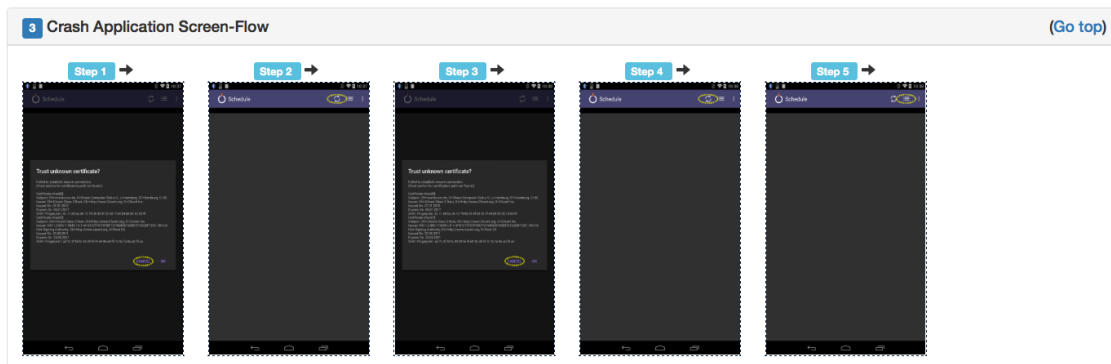


Figure 5.2: Crash Screen-Flow

Android OS, a legend of icons that indicate the current contextual state of the app in the reproduction steps, the device, and the screen orientation and resolution when the crash occurred; 2) natural language sentences that describe the steps to reproduce a crash using detailed information about the GUI events and contextual states for each step (Figure 5.3); 3) an app’s screen flow that highlights the component interacted with on each screen in the execution scenario for a particular crash (Figure 5.2); (4) a pruned stack trace containing only the app exceptions that occurred during execution.

The natural language reproduction steps are constructed by the *Report Generator* (Figure 5.1-④) using the template:

```
<action> on <component text> <component type>, which is located on the <relative
location> of the screen
```

For the steps that have text entry associated with them, the `<action>` placeholder is modified into the following: “Type `<text input>` on the...” so as to capture any specific text inputs that may trigger a crash.

5.2.6 Generating & Replaying Reproduction Scripts

The *Crash Script Generator* (Figure 5.1-⑥), parses the saved execution information from the CrashScope database and generates replayable scripts containing `adb input` commands for touch and text inputs and markers for changes in contextual states. The scripts are

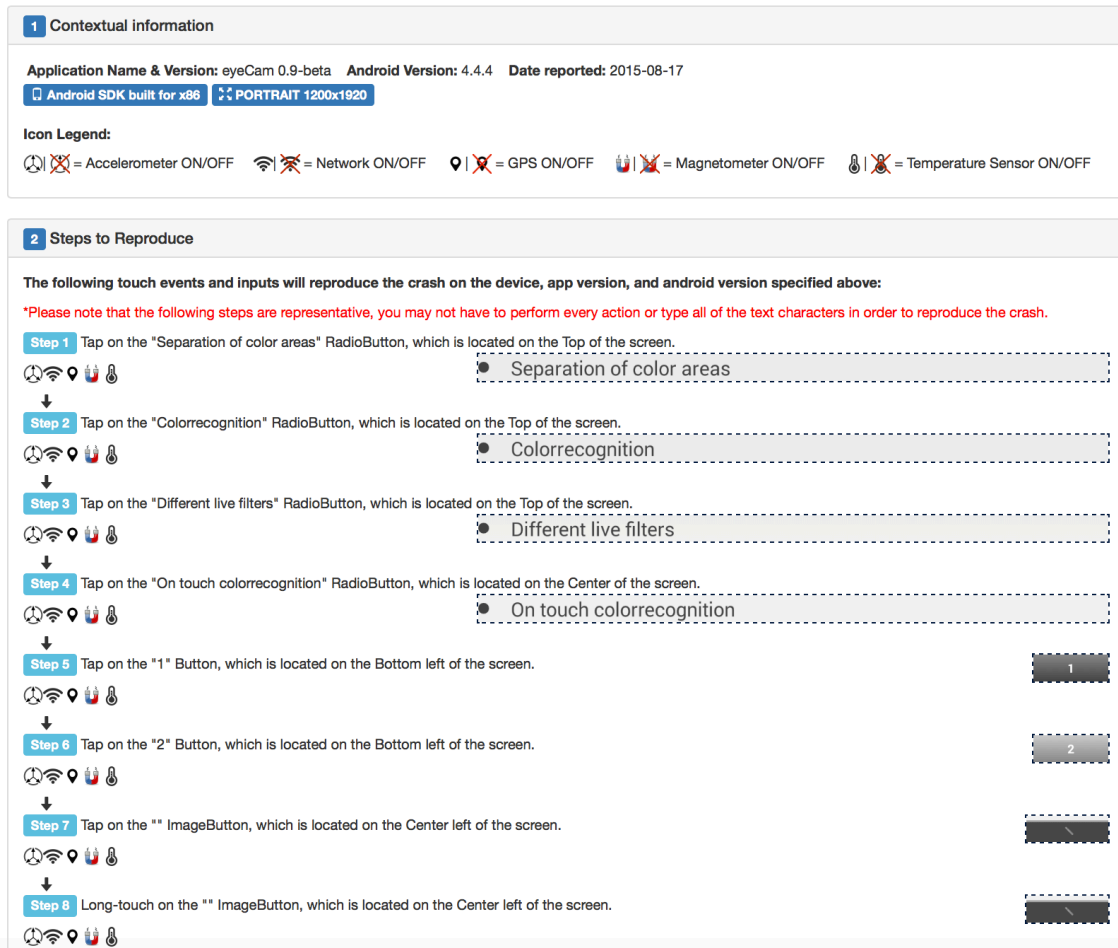


Figure 5.3: Example of Contextual Information and Reproduction Steps sections in a generated crash report

generated by parsing the database for all of the GUI events associated with each step in a particular execution. Then, the coordinates of each component that were recorded during the systematic exploration of the app are parsed and the center coordinates are extrapolated based on each components size. These coordinates are used to generate `adb input` commands to reproduce the GUI event. This approach relies on our previous work in replaying events of test sequences in Android apps [199, 203]. An example of a CrashScope replayable script can be seen in Fig. 5.1-6. The scripts can be replayed by the *Script Replayer* (Fig. 5.1-7), which executes the `adb input` commands, and interprets the

Table 5.1: Tools used in the comparative fault finding study

Tool Name	Android Version	Tool Type
Monkey	any	Random
A ³ E Depth-First	any	Systematic
GUIRipper	any	Model-Based
Dynodroid	v2.3	Random-Based
PUMA	v4.1+	Random-Based

state change markers in the script (*e.g.*, `<Wifi_OFF>`) to execute proper *telnet* commands to set states on an emulator.

5.3 Empirical Study 1: Crash Detection Capability

The *goal* of our first study is to evaluate the effectiveness of CrashScope at discovering crashes in Android apps as compared to state-of-the-art approaches for testing mobile apps. The *quality focus* of this first study concerns the fault detection capabilities of CrashScope in terms of locating crashes. The *context* of this study consists of 61 open-source Android apps previously used to evaluate automated testing approaches in [142], as well as five approaches for automated input generation (listed in Table 5.1). We investigated the following research questions (RQs):

- **RQ₁:** *What is CrashScope’s effectiveness in terms of detecting application crashes compared to other state-of-the-art Android testing approaches?*
- **RQ₂:** *Does CrashScope detect different crashes compared to the other tools?*
- **RQ₃:** *Are some CrashScope execution strategies more effective at detecting crashes or exceptions than others?*
- **RQ₄:** *Does average application statement coverage correspond to a tool’s ability to detect crashes?*

5.3.1 Methodology

In order to compare CrashScope against other state of the art automated input generation tools for Android, we utilized a subset of subject apps and tools available in the Androtest testing suite [142, 16]. We chose to perform this study on a subset of the tools offered by Androtest artifact due to runtime issues, namely, some tools would not run consistently on the set of provided subject apps (*e.g.*, the tools would launch an emulator but not the app), causing inconsistent results we chose to exclude. However, when contacted, the authors of the tool were helpful in supporting us. We believe the tools tested against constitute a diverse representation of the publicly available Android testing tools. The Androtest suite contains 68 subject applications for testing; however, when recompiling the applications to run the tools and extract the apps from the VM to run with CRASHSCOPE, seven of the subject apps failed to compile with the instrumentation necessary to gather code-coverage results. Therefore, each tool in the suite was allowed to run for one hour for each of the remaining 61 subject apps, five times, whereas we ran all 12 combinations of the CrashScope strategies once on each of these apps. It is worth noting that the execution of tools in the Androtest suite (except for Android monkey) can not be controlled by a criteria such as maximum number of events.

In the Androtest VMs, each tool ran on its required Android version, for CrashScope each subject application was run on an emulator with a 1200x1920 display resolution, 2GB of RAM, a 200 MB Virtual sdcard, and Android version 4.4.2 JellyBean. We ran the tools listed in Table 5.1, except Monkey, using Vagrant [106] and VirtualBox [107]. The Monkey tool was run for 100-700 event sequences (in 100 event deltas for seven total configurations) on an emulator with the same settings as above with a two-second delay between events, discarding trackball events. Each of these seven configurations was executed five times for each of the 61 subject apps, and every execution was instantiated with a different random seed [13]. While Monkey is an available tool in Androtest, the authors of the tool chose to set no delay between events, meaning the number of events monkey executed over the

course of 1 hour far exceeds the number of events generated by the other tools, which would have resulted in a biased comparison to the CrashScope and the other automated testing tools. In order to give a complete picture of the effectiveness of CrashScope as compared to the other tools, we report data on both the statement coverage of the tools as well as crashes detected by each tool. Each of the subject applications in the Androtest suite was instrumented with the Emma code coverage tool [40], and we used this instrumentation to collect statement coverage data for each of the apps. Due to space limitations, we report the cumulative coverage for all of the strategies and runs of each tool with a full dataset of detailed statistics available in our replication package in the online appendix [35].

The underlying purpose of this study is to compare the crash detection capabilities of each of these tools and answer **RQ**₁. However, we cannot make this comparison in a straightforward manner. CrashScope is able to accurately detect app crashes by detecting the standard Android dialog for exposing a crash (*e.g.*, a text box containing the phrase “application_name has stopped”). However, because the other analyzed tools do not support identifying crashes at runtime, there is no reliable automated manner to extract instances where the application crashed purely from the `logcat` [9]. To obtain an approximation of the crashes detected by these tools, we parsed the `logcat` files generated for each tool in the Androtest VMs. Then, we isolated instances where exceptions occurred containing the `FATAL EXCEPTION` key marker, which were also associated with the process id (pid) of the app running during the `logcat` collection. While this filters out unwanted exceptions from the OS and other processes, unfortunately, it does not guarantee that the exceptions signify a crash caused by incorrect application logic. This could signify, among other things, a crash caused by the instrumentation of the controlling tool. Therefore, in order to conduct a consistent comparison to CRASHSCOPE, the authors manually inspected the instances of fatal exception stack traces returned by the `logcat` parsing, discarding duplicates and those caused by instrumentation problems, and we report the crash results of the other tools from this pruned list. A full result set with both full and pruned `logcat` traces is available in our online appendix [35]. The issues encountered when parsing the

Table 5.2: Unique Crashes Discovered with Instr. Crashes in parentheses

App	A ³ E	GUI-Ripper	Dynodroid	PUMA	Monkey (All)	CrashScope
A2DP Vol	1	0	0	0	0	0
aagtl	0	0	1	0	1	0
Amazed	0	0	0	0	1	0
HNDroid	1	1	1	2	1	1
BatteryDog	0	0	1	0	1	0
Soundboard	0	1	0	0	0	0
AKA	0	0	0	0	1	0
Bites	0	0	0	0	1	0
Yahtzee	1	0	0	0	0	1
ADSDroid	1	1	1	1	1	1
PassMaker	1	0	0	0	1	1
BlinkBattery	0	0	0	0	1	0
D&C	0	0	0	0	1	0
Photostream	1	1	1	1	1	0
AlarmKlock	0	0	1	0	0	0
Sanity	1	1	0	0	0	0
MyExpenses	0	0	1	0	0	0
Zooborns	0	0	0	0	0	2
ACal	1	2	2	0	1	1
Hotdeath	0	2	0	0	0	1
Total	8 (21)	9 (5)	9 (6)	4 (0)	12 (1)	8 (0)

results from these other tools further highlight CrashScope’s utility, and the need for an automatic tool that can accurately detect and in turn effectively report crashes in mobile apps.

5.3.2 Results & Discussion

Table [5.2](#) shows the aggregated crash discovery results of each tool over their various runs. This table reports unique crashes (as signified by differing stack traces not caused by app instrumentation) detected by the various approaches, only includes those apps for which crashes were discovered. For tools other than CrahsScope, we also report crashes (in parentheses) that were caused by instrumentation frameworks (*e.g.*, troyd, Android intrs., junit, Emma), as these represent “false positive” crashes uncovered by the tools. The results highlight four key results. The first observable result is that CrashScope is about as effective in terms of number of crashes detected, while also providing detailed bug reports. CrashScope discovered fewer crashes compared to Monkey due to the large number of events that this tool is capable of producing. However, it should be noted that

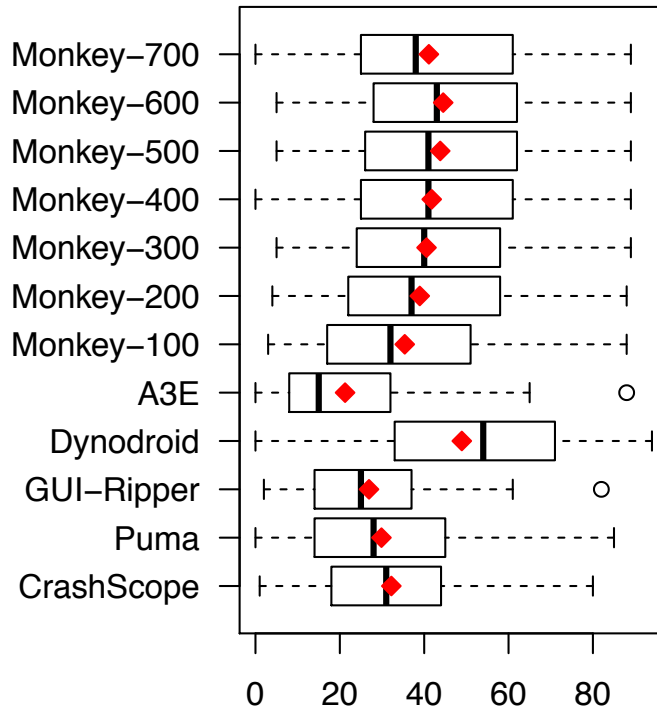


Figure 5.4: Average Coverage Results for the Comparative Study

Monkey is not able to generate replayable scripts or reports, severely limiting its usefulness from a developers perspective. CrashScope was able to discover about as many crashes as A³E, GUI-Ripper, and Dynodroid, more than PUMA, without any false positives caused by instrumentation of the app or system. Therefore, we answer **RQ₁** as follows: **CrashScope is about as effective at detecting crashes as the other tools. Furthermore, our approach reduces the burden on developers by reducing the number of “false” crashes caused by instrumentation and providing detailed crash reports.**

The second observable result is that CrashScope is able to detect orthogonal crashes compared to the other tools. In order to understand why CrashScope detected different crashes than the other approaches, the authors manually examined the detected crash reports to determine their causes. Because it might not be possible to determine the exact cause or type of crashes from the other tools, we exclude a discussion here, but we speculate on the differences from CrashScope’s results. The key finding from this exploration

is that *the differing strategies implemented by CrashScope contributed to its ability to detect orthogonal crashes compared to the other tools*. For instance, the crash detected by CrashScope for the *zooborns* app is triggered by typing unexpected text in a text box. The other tools probably missed this crash because their text generation techniques do not include unexpected inputs. Furthermore, one aspect of this crash highlights the utility of CRASHSCOPE’s detection and reporting capabilities, namely, the thrown exception is potentially misleading to a developer. While this crash was caused by text formatting, the exception is for an AsyncTask object, one of Android’s thread handling mechanisms, meaning it could be difficult for a developer to reason about the cause of this crash in the absence of a detailed report. Another example of an orthogonal crash discovered by CrashScope is that for the *PasswordMakerPro* app. While two other tools (Monkey, A³E) found a crash during their exploration of this app, only CrashScope was able to discover a crash caused by a contextual feature, rotation. This highlights the utility of the different exploration techniques. Consequently, **RQ₂** can be answered as follows: **The varying strategies of CrashScope allow the tool to detect different crashes compared to those detected by other approaches.**

The third result we see from the the crash detection data is that certain CrashScope strategies are more effective at uncovering crashes than others. The most effective of the text strategies overall was the *unexpected* heuristic that was able to discover all of the crashes listed for CrashScope in Table 5.2. Different crashes were discovered during the runs of strategies where contextual features were and were not tested in adverse conditions, as discussed above, suggesting that some errors are only discoverable when contextual features are in normal states. Overall, the *forwards* heuristic for traversing the GUI led to the discovery of more crashes (8 crashes) compared to the *backwards* strategy (7 crashes), with some of these crashes overlapping. The most effective overall crash discovery strategy was $\langle \textit{contextual_feautres_enabled}, \textit{forward}, \textit{unexpected} \rangle$. Thus, **RQ₃** can be answered as **Different combinations of CrashScope strategies were more effective than**

others, suggesting the need for multiple testing strategies encompassed within a single tool.

The fourth observable result is that the average statement coverage of the analyzed tools tool (see Fig. 5.4) does not necessarily correspond to a better fault discovery capability, as CrashScope was able to detect about as many crashes with lower average coverage than other tools (*i.e.*, PUMA, Monkey, and Dynodroid). This implies that future testing approaches for mobile apps need to take into consideration metrics in addition to code coverage to illustrate the effectiveness of the approach. Therefore, our answer for **RQ₄** is: **Higher statement coverage of an automated mobile app testing tool does not necessarily imply that tool will have effective fault-discovery capabilities.**

5.4 Study 2: Reproducibility & Readability

The *goal* of the second study is to evaluate the *reproducibility* and *readability* of the natural language reports generated by CrashScope compared to original human written reports found in online issue trackers. The *quality focus* of this study concerns the ability of developers to reproduce bugs from CRASHSCOPE's reports. The *context* of this study consists of eight real world Android app crashes and reports, extracted from open source apps and their corresponding issue trackers, as well as reports generated by CrashScope for these same crashes (details of the crashes and corresponding apps are presented in our online appendix 35). In the context of this second study we examined the following RQs:

- **RQ₅**: *Are reports generated with CrashScope more reproducible than the original human written reports?*
- **RQ₆**: *Are reports generated by CrashScope more readable than the original human written reports?*

Table 5.3: User Experience Results: This table reports the mean average response from 16 users regarding the User Experience questions posed for both CrashScope generated reports and the original human written reports found in the app’s issue trackers. (CS = CrashScope Bug Reports, O=Original Bug Reports, M =Mean, SD =Standard Deviation)

Question	CS M	CS SD	O M	O SD
UX1: I think I would like to have this type of bug report frequently.	4.00	0.89	3.06	0.77
UX2: I found this type of bug report unnecessarily complex.	2.81	1.04	2.125	0.96
UX3: I thought this type of bug report was easy to read/understand.	4.00	0.82	3.00	0.97
UX4: I found this type of bug report very cumbersome to read.	2.50	1.10	2.44	0.81
UX5: I thought the bug report was really useful for reproducing the crash.	4.13	0.62	3.44	0.89

5.4.1 Methodology

To identify the crashes used for this study, we manually inspected the issue trackers of the apps on F-droid looking for reports that described an app crash. Then, we ran CrashScope on the version of the app that the crash was reported against to observe whether or not CrashScope was able to capture the crash on the same emulator configuration as the previous study. While we chose these bugs manually, the goal of this study is not to measure CRASHSCOPE’s effectiveness at discovering bugs (unlike the first study). We acknowledge that there are situations in which CrashScope will not be able to detect a fault and we outline these cases in Section 5.5

In order to answer **RQ**₅ and **RQ**₆, we asked 16 CS graduate students from William and Mary (a proxy for developers [247]) to reproduce the eight crashes (four from the original human written reports, and four from CRASHSCOPE). The design matrix of this study was devised in such a way that each crash for each type of report was evaluated by four participants, no crash was evaluated twice for the same participant, and eight participants saw the human written reports first, and eight participants saw the CrashScope reports first, all in the interest of reducing bias. The system names were also anonymized (CrashScope to “System A” and the human written reports to “System B”). The full design matrix can be found in our online appendix [35]. During the study, participants recorded the time it took them to reproduce the crash on a Nexus 7 device for each report, with a time limit of ten minutes for reproduction. If a participant could not reproduce the bug within the

ten minute time frame or gave up in trying to reproduce the bug, that bug was marked as non-reproducible for that participant. Therefore, in order to answer **RQ₅**, we measured how many crashes were successfully reproduced by the participants for each type of crash report, we also measured the time it took each participant to reproduce each bug (the detailed dataset is available at [35]).

After the completion of the crash reproductions, we had each participant fill out a brief survey, answering questions regarding the *user preferences* (**UP**) and *usability* (**UX**) for each type of bug report. We also collected information about each participants programming experience and familiarity with the Android platform. The **UP** questions were formulated based on the user experience honeycomb originally developed by Moville [229] and were posed to participants as free form text entry questions. We objectively measure the *user preferences* of the participants by summarizing the responses and offering excerpts from the answers highlighting the results. The **UX** questions were created using statements based on the SUS usability scale by Brooke [132] and were posed to participants in the form of a 5-point Likert scale. We quantify the *user experience* of CrashScope and answer **RQ₆** by presenting the mean and standard deviation of the scores for the responses to the Likert-based questions. The questions regarding programming experience are based on the questionnaire developed by Feigenspan *et al.* [156].

5.4.2 Results & Discussion

The CrashScope reports achieved a similar levels of reproducibility compared to the human written reports with 94% (60 out of 64) of the CrashScope reports being successfully reproduced by participants compared to 92% (59 out of 64) of the original reports. Therefore, **RQ₅** can be answered as follows: **Reports generated by CrashScope are about as reproducible as human written reports extracted from open-source issue trackers.** The UX questions and results can be found in Table [5.3] which show that participants found CrashScope reports to be more readable and useful than the original reports. Thus, **RQ₆** can be answered as: **Reports generated by CrashScope are more readable**

and useful from a developers' perspective as compared to human written reports. One interesting case arose from this study. No participant assigned the original report for the C13C Schedule app was able to reproduce the bug, whereas all participants assigned the CrashScope version of this app were able to reproduce it. This is because the network needed to be disabled for the crash to manifest itself, and this was not captured in the original bug report. This highlights the utility of CrashScope's context-aware reports.

5.5 Limitations & Threats to Validity

While our empirical evaluation has shown that CrashScope is effective at detecting crashes in Android apps, our tool has some inherent limitations. *First*, because CRASHSCOPE's systematic execution engine does not implement the swipe gesture, it will not be able to execute GUI components existing within a list that does not fit entirely within the device's screen. This limitation may cause some crashes or exceptions dependent on these types of components to be missed. The *second* limitation is that CrashScope does not support highly specialized text input. This may limit the exploration capabilities of our tool for certain apps. However, recent approaches in concolic and symbolic executions may prove useful in overcoming this limitation [252, 213, 174, 271]. The *third* limitation of our tool relates to window detection in Android. Android apps are organized into screens based on activities and other windows (*e.g.*, dialogs). Activities are fairly simple to detect, as each has a unique name which acts as an identifier for that activity. However, the same is not true for dialogs, as they have no unique identifier. Each Activity can have multiple dialogs. To solve this problem we use the size of the window with the focus and in foreground as a unique identifier, as through our observations we found that very few activities employ different unique windows of the same size. However, this is an imperfect heuristic and prone to occasional errors. Due to checks in place in our systematic execution algorithm, this never leads to incorrect execution of the app, however, it may mean that less functionality

of the app is explored compared to a method that is able to correctly identify all unique windows in an app.

One potential threat to external validity is the fact that we used a set of 61 open source applications to evaluate CRASHSCOPE in the first empirical study, and eight crashes in eight open source applications for the second empirical study. Therefore, we can not generalize our results to Android apps in general due to the limitations of these subject apps. However, we believe that this threat is lessened by the fact that these apps were collected from datasets in previous studies and contain several popular, complex apps. In the context of our empirical studies, one threat to internal validity stem from the potentially surprising effects of participants in the user study for the second empirical study. To this end there is a threat since we approximated graduate students in Computer Science as experienced Android developers. However, this threat is mitigated by the fact that all of these participants indicated that they have extensive programming experience as well as moderate experience with the Android environment, and recent work shows that in carefully controlled experiments experienced graduate students are sufficient proxy's for developers [247]. Another threat to internal validity concerns the manual inspection of log traces from the tools CrashScope was tested against. However, this threat is mitigated due to the fact that the the process was partially automated to decrease the manual examination set and the authors who examined these logs are very well versed in the Android platform and automated testing approaches in research.

5.6 Conclusions

In this chapter, we presented CRASHSCOPE, a practical approach for discovering, reporting, and replaying Android app crashes. Our tool leverages a powerful algorithm for systematic exploration that is crash tolerant, capable of context-aware input and text generation, and runs on a diverse set of devices and emulators. We evaluated CrashScope with respect to crash and exception detection, as compared to other state-of-the-art automatic input

generation tools for Android and show that our tool is able to uncover at least as many crashes as these other approaches, while offering more detailed information in the form of NL crash reports containing steps to reproduce the crash, and high-level repayable traces that can reproduce the crash on demand. We also evaluated the *reproducibility* and *readability* of our automatically generated reports and show that they provide for reliable reproduction of crashes while proving more readable and usable for developers. In the future, we aim to investigate techniques to trim bug reports, so that they contain only the necessary steps, as well as improving our systematic exploration strategy for uncovering a higher number of bugs, by adapting promising emerging approaches in model-based GUI testing. [233].

Chapter 6

Conclusions & Future Research

In this dissertation, we have presented several different approaches for automating the software development process of mobile applications and both empirical validations of their effectiveness and evidence of their applicability to real development workflows. More specifically, we have helped to automate various aspects of the design, implementation, and testing of apps. However, the presented work only touches the surface of various components of the development process that are ripe for automation. Thus, there are several promising avenues of future work related to software automation for mobile applications. In this chapter, we outline three of these topics before offering concluding remarks on the techniques presented in this dissertation. These three topics include (i) providing automated documentation related to graphical user interfaces, (ii) helping to improve program comprehension and enable practical program synthesis by leveraging information encoded into graphical user interfaces, and (ii) working towards a new vision for automated mobile testing centered around three principles: *continuous*, *evolutionary*, and *large-scale* [197]. In our discussion of these directions for future work, we continue our focus upon the domain of mobile apps. However, many of the underlying principles of the work discussed are transferrable to other domains, particularly those concerned with GUI-centric applications.

6.1 Toward Automatically Documenting Graphical User Interfaces

Highly competitive app stores like Apple’s App Store [24] or Google Play [50] contain millions of apps, many of which implement similar functionality. In order to succeed in such marketplaces, developers need to ensure their application provides an engaging user experience and aesthetically pleasing user interface [112]. Unfortunately, past studies have shown that designing and implementing effective GUIs can be a difficult task [259, 231, 232], especially for mobile apps [224]. These difficulties are due in large part to challenges unique to the mobile development process that have been well documented in research literature [180] and include: (i) rapidly evolving platforms and APIs [200, 125], (ii) continuous pressure for new releases [170, 178], (iii) inefficiencies in testing [142, 196, 197], (iv) overwhelming and noisy feedback from user reviews [144, 152, 238, 237], and (v) market, device, and platform fragmentation [165, 264, 7].

Mobile GUIs are typically stipulated in files separate from the main logic of the app (*e.g.*, `.xml` for Android, and `.nib` or storyboards for iOS). These files delineate attributes of GUI components in relative terms (*e.g.*, display independent pixel `dpi` values) and are arranged according to a hierarchical structure (*i.e.*, a GUI hierarchy) to facilitate reactive design across fragmented device ecosystems. Reasoning about the actual rendering of a GUI using such an abstract definition in code is a difficult task. Conversely, collecting screenshots to discern visual changes is difficult, as it requires manual intervention and adept visual perception is needed to discern meaningful GUI changes. Thus, it is clear that comprehending how GUI code affects the visual representation of an app requires mentally bridging a challenging abstraction gap.

Furthermore, the design and implementation of a GUI for a mobile app is not a “single cost” task that is performed at the inception of development. Instead, GUI-changes must evolve to keep pace with constant user feedback and the evolution of the prescribed design language and guidelines of the underlying mobile platform (*e.g.*, Android’s transitions

to differing versions of material design [49]), thus developers must constantly evolve an app's GUI to satisfy changing design requirements. This illustrates that there is a clear need for automated support in effectively *documenting* GUI changes to help aid developers in time-consuming program comprehension tasks related to mobile app development. In particular, automated summarization of *visual* GUI-changes would allow for developers to more effectively comprehend the affect of code-based changes on the visual representation of a mobile GUI.

It is clear that automated support in documenting graphical user interfaces for mobile apps would greatly benefit developers. The overarching goals of this future research thrust regarding automated GUI documentation are as follows:

- **Research Goal 1:** *Understanding Developer's and User's Information Needs in Documenting GUIs:* In order to create effective automated documentation for graphical user interfaces, it is important to first understand what documentation information both developers and users find useful. Thus, the first goal of this research thrust is to conduct studies that will shed light on information needs for GUI documentation.
- **Research Goal 2:** *Designing Developer and User-Centric Approaches for Automated GUI Documentation:* Once we have established a set of guidelines for effective GUI documentation in eyes of developers and users, we will leverage this knowledge to create approaches that are capable of automatically documenting GUIs as software evolves.

6.2 Toward GUI-centric Automated Program Understanding & Synthesis

The Graphical User Interfaces of software applications contain a wealth of information that may be useful for aiding in automated program understanding, and in the future, program

synthesis. In this dissertation, we have illustrated an effective and promising approach for automated synthesis of code that implements a specified graphical user interface of a mobile app. However, this is only the first step toward a more complete process of program synthesis. Given recent advancements in artificial intelligence and machine learning techniques, particularly as they relate to computer vision, one could conceive of moving beyond the capabilities of REDRAW, towards implementing different functional properties of a GUI. However, to accomplish this, the extent to which the visual semantics of graphical user interfaces can encode underlying functional information of a software GUI must be thoroughly explored.

The overarching goals of this future research thrust in program understanding and synthesis are as follows:

- **Research Goal 1:** *Explore the Representational Power of Graphical User Interfaces:* In order to move toward approaches capable of automatically generating functional GUI-related code, the degree to which this functional information can be learned from GUIs must be explored. In essence, this requires studies focused on ascertaining the representational properties of GUIs as related to software functionality.
- **Research Goal 2:** *Designing Approaches for Synthesizing functional GUI-related Code:* According to the information gleaned from studying the representational power of GUIs, we will design approaches for synthesizing code related to various discrete functional properties of software GUIs.

6.3 Toward a Practical, Comprehensive Framework for Automated GUI-based Testing

Unique characteristics and emerging best practices for creating mobile apps, combined with immense market interest, have driven both researchers and industrial practitioners to devise frameworks, tools, and services aimed at supporting mobile testing with the goal

of assuring the quality of mobile apps. However, current limitations in both manual and automated solutions underlie a broad set of challenges that prevent the realization of a comprehensive, effective, and practical automated testing approach [180, 142, 185]. Because of this, mobile app testing is still performed mostly manually costing developers, and the industry, significant amounts of effort, time, and money [180, 142, 185]. As development workflows increasingly trend toward adoption of agile practices, and continuous integration is adopted by larger numbers of engineers and development teams, it is imperative that automated mobile testing be enabled within this context if the development of mobile apps is to continue to thrive. However, current solutions for automated mobile testing do not provide a “fully” automated experience, and several challenges are still open issues requiring attention from the community, if the expected goal is to help mobile developers to assure quality of their apps under specific conditions such as pressure from the users for continuous delivery and restricted budgets for testing processes. As part of our proposed future work, we introduce a new paradigm for mobile testing called CEL testing, which is founded on three principles: *Continuous*, *Evolutionary*, and *Large-scale* (CEL).

6.3.1 The CEL Testing Principles

The CEL testing framework is based on three core principles aimed at addressing these challenges: *Continuous*, *Evolutionary*, and *Large-scale*. These principles integrate and extend concepts from software evolution and maintenance, testing, agile development, and continuous integration. However, the principles alone are not enough to provide solutions to the aforementioned challenges. Therefore, as part of the CEL testing vision, we propose a system architecture for automated mobile testing following CEL principles. To make this vision tractable, we propose a research agenda for enabling CEL testing and implementing our envisioned system.

Automated testing of mobile apps should help developers increase software quality within the following constraints: (i) restricted time/budget for testing, (ii) needs for diverse

types of testing, and (iv) pressure from users for continuous delivery. Following the CEL principles can enable effective automated testing within these requirements:

Continuous. Following the principles that support continuous integration and delivery (CI/CD), mobile apps should be continuously tested according to different goals and under different environmental conditions. Tests should simulate real usages and consider scenarios that simulate different contextual eventualities (*e.g.*, exploring a photos app when losing connectivity) as dictated by app features and use cases. Any change to the source code or environment (*i.e.*, usage patterns, APIs, and devices) should trigger — automatically — a testing iteration on the current version of the app. To avoid time-consuming regressions, test cases executed during the iteration should cover only the impact set of the changes that triggered the iteration. Finally, to support practitioners when fixing bugs, the bug reports generated with CEL testing should be expressive and reproducible, *i.e.*, the bug reports should contain details of the settings, reproduction steps, oracle, inputs (GUI and contextual events), and stack traces (for crashes).

Evolutionary. App source code and testing artifacts (*i.e.*, the models, the test cases, and the oracles) should not evolve independently of one another; the testing artifacts should adapt automatically to changes in (i) the app, (ii) the usage patterns, and (iii) the available devices/OSes. Thus, the testing artifacts should continuously and automatically evolve, relying not only on source code changes as an input, but also information collected via MSR techniques from sources such as on-device reporting/monitoring, user reviews, and API evolution. CEL testing employs a multi-model representation of the app and this mined data, consisting of GUI, domain, usage, fault, and contextual models, to properly evolve the testing artifacts. This multi-model representation can be used for the evolutionary generation of testing artifacts which consider *both* historical and current data.

Large-scale. To assure continuous delivery in the face of challenges such as fragmentation, constrained development timelines, and large combinations of app inputs from GUI and contextual events, CEL requires a large-scale execution engine. This engine should enable execution of test cases that simulate real conditions in-the-wild. Therefore, to sup-

port a large test-matrix, CEL testing should be supported on infrastructures for parallel execution of test cases on physical or virtual devices. While virtual devices reduce the cost of implementing the engine, physical devices (or extremely accurate simulations) are mandatory for performance testing. The large-scale engine should be accessible in the context of both cloud and on-premise hardware. Thus, an open-source implementation of the engine is preferred because CEL testing is targeted for both professional development and SE research.

Based on the current frameworks, tools, and services that are available to developers, as well as the limitations and remaining open challenges in the domain of mobile testing, we firmly believe that our vision for *Continuous, Evolutionary* and *Large-Scale* mobile testing offers a comprehensive architecture that, if realized, will dramatically improve the testing process. However, there are still many components of this vision that are yet to be properly explored in the context of research. Therefore, in order to make our vision for the future of mobile testing tractable, we offer an overview of a research agenda broken down into six major topics.

- **Research Goal 1:** *Toward Improved Model-Based Representations of Mobile Apps:* Current approaches for deriving model-based representations of apps are severely lacking a multi-model-based approach that might significantly improve the utility of model-based testing. However, to this end, there are several unexplored areas requiring further research and investigation. While model-based representations of mobile GUIs have been widely explored [120, 141, 273, 268, 203, 278], researchers should focus on unifying the (often complementary) information which can be extracted from both static and dynamic program analysis techniques. For instance, using static control flow information from a tool like GATOR to guide dynamic GUI-ripping to extract a more complete *GUI model*. Very little research work has been devoted to deriving *domain models* from applications, however, such models will be crucial for enabling automated tests to exercise complex inputs and behaviors. Future studies

could focus on automatically extracting domain models from source code and data storage models, and by examining common traits between apps that exist in similar categories in app marketplaces in order to derive common event sequences and GUI-usage patterns.

Given the highly contextualized environment of mobile apps (e.g., varying network and sensor conditions), effective automated testing will require a *contextual model* identifying and quantifying the usages of related APIs within in application. While some recent work has explored such functionality [219], this can be made more precise and robust through more advanced static analysis and dynamic techniques that infer potential context values to help drive automated testing. Very few automated testing approaches for mobile apps consider *usage models* [203, 214] stipulating common functional use cases of an app, expressed as combinations of GUI events. Recent advances in deep-learning based representations may be applicable for appropriately modeling user interactions and high-level features, if properly cast to the problem domain.

In order to better inform test case generation and properly measure the effectiveness of automated testing, platform specific *fault models* must be empirically derived through observations and codification of open source mobile app issue trackers, and knowledge bases such as Stack Overflow [88] or the XDA developer forums [116]. Finally, in order for these models to be viable within an evolutionary context, there must exist mechanisms for accurate, history aware model updates. A continuously evolving model will allow for more robust updates to generated test-related artifacts.

- **Research Goal 2:** *Goal-Oriented Automated Test Case Generation:* Current approaches for automated input generation for mobile apps have typically focused on a single type of testing, namely *destructive testing* [39] or some derivation thereof. The effectiveness of such techniques are typically measured code coverage metrics or by the number of failures uncovered. While this type of testing can help improve the

quality of an app, it is one of *many* important testing practices in the mobile domain. In order to provide developers with a *comprehensive* automated testing solution, researchers must focus on automated test generation for other types of testing aimed at different goals, particularly those measuring mobile-specific quality attributes. Some of these testing types include security testing, localization testing, energy testing, performance testing and play-testing. Testing for different goals on mobile platforms *fundamentally differs* from similar testing scenarios for other types of software due to the GUI and event-driven nature of mobile apps, and the fact that GUI tests on devices are currently a necessity (as unit testing misses important features untestable outside of device runtimes) for exercising enough app functionality to achieve effective practices for many of these testing scenarios. Therefore, the challenge to the research community is to utilize the representation power of the models we describe in this paper to devise techniques for automated test case generation for different *testing goals*.

- **Research Goal 3:** *Flexible Open Source Solutions for Large Scale and CrowdSourced Testing:* As mobile markets mature and additional devices are introduced by consumer electronics companies, the mobile fragmentation problem will only be exacerbated. As previously discussed, cloud-based services offering virtually accessible physical devices and crowdsourced testing are two promising solutions to this issue, however, these solutions are not available to all developers and are not scalable to all testing goals. For instance, it may be difficult to carry out effective energy or security testing on cloud-based devices if such services are not specifically enabled by a cloud provider. As outlined in our vision, we looked to container and virtual machine technology that has made testing practices scalable in development scenarios like continuous integration (CI). Thus, it is clear that a robust and highly customizable container or virtualization image of a mobile platform is the most promising long-term, scalable solution for enabling our vision of CEL testing. Future research in the

systems area could focus on improving the viability of promising open source source projects as androidx86 [15] to be used in CI-like development environments, allowing for further customizations and control over attributes such as sensor value mocking and screen size and pixel density. While valuable, these virtual devices will not be applicable to all types of testing, such as usability testing, or usage information collection which can be used to derive an effective usage model of an app. Instead, such goals fit the model of crowdsourced testing well. Unfortunately, no flexible open source solutions to support developers or researchers currently exist, signifying the need for such a platform. Luckily, there are existing modern open source solutions such as OpenSTF [90] and ODBR [223] that could help facilitate the creation of such a platform. This platform should allow for easy collection of privacy-aware execution traces and logs, suitable for deriving usage models.

- **Research Goal 4:** *Derivation of Scalable, Precise Automated Oracles:* To allow viable automated support of a diverse set of testing goals, progress must be made in the form of automatically generated, accurate, and scalable oracles. It is likely that such oracles will be specific to particular types of testing tasks and require different technological solutions. Some automated testing approaches have broached this problem and devised simple solutions such as using app agnostic oracles based on screen rotation actions [273] or GUI screenshots as state-representations [195]. However, there are still open problems even with these simple types of oracles, and they are not comprehensive. Promising directions along this research thread might include mixed GUI representations that utilize both image and textual representations of GUI information to form robust state indications, which could be used as automated oracles. Additionally, the derivation of mobile platform-specific fault models may help in deriving automated oracles that could test for common problems inherent to mobile apps.

- **Research Goal 5:** *Mining Software Repositories and User Reviews to Drive Testing:* While many different automated testing solutions for mobile apps have been proposed, they largely ignore information sources which could be invaluable for informing the testing process, namely data mined from software repositories and user reviews. Information from software repositories for mobile apps could be collected in two ways, which could be combined to maximize the information utility, (i) mining the development history of a single application, and (ii), the development history of collections of open source apps hosted on services like GitHub. Here lightweight static analysis techniques could be used at scale, whereas more expensive app control flow analysis techniques could be used to provide more detailed code-level information about a single subject app. Mobile app developers also have an unprecedented feedback mechanism from users in the form of user reviews. As such there is a growing body of work that has focused on identifying informative reviews [140, 237, 261], linking these to affected areas of source code [237], and even recommending code changes [238]. However, little work has been done to use the information contained within informative reviews to drive different types of testing. For instance, in the context of functional or regression testing, user reviews could be used to prioritize test cases, or even generate test cases for issues derived from reviews.
- **Research Goal 6:** *Derivation of Methods to Provide Useful Feedback for Developers:* In order to make the results of automated testing practices *useful* and *actionable* for developers, researchers must dedicate effort to (i) deriving useful visual representations of testing results, and (ii) augmenting typical methodologies by which users might report feedback to developers. Very few automated testing approaches have considered methodologies for augmenting or effectively reporting testing information to developers [225, 219]. Here researchers might consider applications of promising visualization approaches adopted from the HCI community combined with developer information needs derived from empirical studies. The studies conducted with engi-

neers can help to develop theoretically grounded solutions for providing them with actionable information and augmented context (*e.g.*, sound traceability links back to different parts of application code). Additionally, novel mechanisms for aiding users in providing actionable feedback to developers will be important to increase the quality of mineable information (*e.g.*, on-device bug reporting and monitoring).

6.4 Concluding Remarks

We opened this dissertation with a thesis statement that asserted in essence, that *automating the process of implementing and reasoning about code from abstract concepts will lead to more effective, and more efficient software development practices*. While the work that has been presented over the course of this document only begins to investigate the extent to which this assertion is true, there is not doubt that the results are promising. To summarize our contributions, in this dissertation we presented three novel approaches that automated the software design and testing processes for mobile apps. First, we introduced GVT, that is capable of resolving instances where the implementation of a mobile application’s GUI does not meet its intended specifications. Second, we introduced REDRAW, a technique for automatically generating GUI-related code for a mobile application taking only a screenshot as input. Finally, we introduced CRASHSCOPE, which is capable of automatically performing GUI-based testing of mobile apps, detecting crashes, and producing expressive, useful crash reports.

We offer a combination of both quantitative empirical evidence, and qualitative evidence collected from user studies with professional developers which supports our core thesis that automation can improve the processes of designing and implementing software. First, to evaluate GVT we carried out both a controlled empirical evaluation with open-source applications as well as an industrial evaluation with designers and developers from Huawei, a major software and telecommunications company. The results show that GVT is able to detect and report violations of GUI design specifications with remarkable efficiency

and accuracy and is both useful and scalable from the point of view of industrial designers and developers. GVT’s industrial applicability is bolstered by the fact that, at the time of this dissertation’s publication, over one-thousand industrial designers and developers at Huawei actively utilize our approach to improve the quality of their mobile apps. Second, our evaluation of REDRAW illustrates that our approach’s CNN achieves an average GUI-component classification accuracy of 91% and assembles prototype applications that closely mirror target mock-ups in terms of visual affinity while exhibiting reasonable code structure. Furthermore, interviews with industrial practitioners from Google, Facebook, and Huawei illustrate REDRAW’s potential to improve real design and development workflows. Finally, we evaluated CrashScope’s effectiveness in discovering crashes as compared to five state-of-the-art Android input generation tools on 61 applications. The results demonstrate that CRASHSCOPE is able to uncover crashes that other tools failed to detect and provides more detailed fault information. Additionally, in a study analyzing eight real-world Android app crashes, we found that CrashScope’s reports are easily readable and allow for reliable reproduction of crashes by presenting more explicit information than human written reports. While there is still much work to be done, this evidence helps support the notion that practical applications to software engineering processes can dramatically improve the effectiveness and efficiency of developers.

Appendix A

Individual Contributions to Projects

A.1 Individual Contributions to the GVT Project

- ***Kevin Moran:*** Kevin was the lead researcher who drove the conceptualization and development of the GVT approach. He designed the overall architecture of the approach, formulated the experimental investigation, and wrote the paper. He also lead the implementation of GVT, in particular the components related to the graphical user interface of the tool and the computer vision techniques. He also formulated and generated the material related to user study with developers and professionals at Huawei. Kevin also participated in weekly meetings with professionals and researchers at Huawei to review project progress and guide implementation towards specifications from designers and developers.
- ***Boyang Li:*** Boyang worked primarily on the implementation of the GVT tool, in particular on the components that parsed and matched GUI-metadata from the mock-up metadata and app implementation. Boyang assisted in designing these components of the approach in combination with Kevin. He also assisted in carrying out the survey with professionals at Huawei and translated the the survey and responses between English and Chinese. Boyang assisted in revising the paper and participated in weekly meetings with professionals and researchers at Huawei to review project

progress and guide implementation towards specifications from designers and developers.

- ***Carlos Bernal Cardenas:*** Carlos worked primarily on enabling the injection of the synthetic design violations for the empirical study of GVT’s performance. He also helped to implement the component of the tool that parsed and manipulated the GUI-metadata from the app implementations. Carlos aided in paper revisions and also helped to manually verify some of the mockups utilized in the empirical evaluation of GVT.
- ***Dan Jelf:*** Dan worked primarily on enabling the empirical study conducted to evaluate GVT by reverse engineering the Sketch mock-ups utilized. He also helped to run GVT on all subject screens for the study and to calculate the study metrics based on GVT’s output. Dan aided in paper revisions.
- ***Denys Poshvanyk:*** Denys served as the faculty advisor on this project and helped to guide the conceptualization and evaluation of the GVT approach. He was also heavily involved in revising the paper and participated in weekly meetings with professionals and researchers at Huawei to review project progress and guide implementation towards specifications from designers and developers.
- ***Other Acknowledgements*** – The authors would like to thank Kebin Xie and Roozbeh Farahbod from Huawei’s European Research Center in Munich for their guidance and collaboration with regard to the industrial components of this project. The authors also thank all of the developers and designers at Huawei who helped to pilot GVT and gave valuable feedback about the tool.

A.2 Individual Contributions to the REDRAW Project

- ***Kevin Moran:*** Kevin was the lead researcher who drove the conceptualization and development of the REDRAW approach. He designed the overall architecture of the

approach, formulated the experimental investigation, and wrote the paper. He also lead the implementation of REDRAW, in particular the components related to the convolutional neural network and computer vision techniques. He also formulated and conducted the semi-structured interviews with developers and professionals at Google, Huawei, and Facebook.

- ***Carlos Bernal Cardenas:*** Carlos worked primarily on adapting the systematic exploration approach to perform a large-scale exploration of app GUIs in an efficient manner in order to derive the initial dataset for REDRAW. He also aided in filtering undesirable screens and apps from this dataset using static analysis, and helped to calculate the image similarity metrics for the user study.
- ***Michael Curcio*** Michael worked primarily on three project aspects: (i) training and tuning the convolutional neural network, (ii) the implementation of the K-nearest neighbors approach for constructing the GUI-hierarchy, and (iii) the component that translates a GUI-hierarchy into compilable and runnable GUI-code. He also helped to conduct the empirical evaluation of REDRAW and aided in revising the paper.
- ***Richard Bonett:*** Richie worked primarily on setting up and running the re-implementation of the REMAUI approach for the empirical evaluation.
- ***Denys Poshvanyk:*** Denys served as the faculty advisor on this project and helped to guide the conceptualization and evaluation of the REDRAW approach. He was also heavily involved in revising the paper.
- ***Other Acknowledgements*** – The authors would like to thank Steve Walker and William Hollingsworth for their contributions to the re-implementation of the REMAUI technique as part of a class project during a software engineering course at William & Mary. The authors would also like to thank Benjamin Powell, Jacob Harless, Ndukwe Iko, and Wesley Hatin for their work on translating GUI-hierarchies into

compilable GUI code in the context of a class project during a software engineering course at William & Mary.

A.3 Individual Contributions to the CRASHSCOPE Project

- ***Kevin Moran:*** Kevin was the lead researcher who drove the conceptualization and development of the CRASHSCOPE approach. He designed the overall architecture of the approach, formulated the experimental investigation, and wrote the paper. He also lead the implementation of CRASHSCOPE, in particular the components related to the testing strategies for exercising GUI exploration, text entry, and contextual features. He also formulated and wrote the material related to user study.
- ***Mario Linares-Vasquez:*** Mario primarily worked on the implementation of the crash report generation, helped to design the empirical study for evaluating the CRASHSCOPE approach, and aided in revising the paper.
- ***Carlos Bernal Cardenas:*** Carlos worked primarily on implementing the Android utilities that enabled the extraction and manipulation of GUI-related information from apps running on an Android device or emulator. He also helped to design the initial algorithm for the depth first search-based exploration of a mobile app's GUI along with Kevin. Carlos aided in revising the paper.
- ***Christopher Vendome:*** Chris aided in carrying out the experimental evaluation of CRASHSCOPE by running existing mobile testing approaches and collecting evaluation metrics. Chris also aided in revising the paper.
- ***Denys Poshyvanyk:*** Denys served as the faculty advisor on this project and helped to guide the conceptualization and evaluation of the CRASHSCOPE approach. He was also heavily involved in revising the paper.

A.4 Individual Contributions to the Formulation of CEL Mobile Testing

- **Mario Linares-Vasquez:** Mario formulated the three principles, *continuous*, *evolutionary*, and *large-scale*, upon which the research vision for CEL is based. He also developed the theoretical software architecture of a system that could support CEL testing.
- **Kevin Moran:** Kevin conducted a comprehensive literature review of automated mobile testing approaches which helped lead to the formulation of the CEL principles. Additionally, Kevin developed the concrete research agenda which aims to enable CEL testing via a set of research goals that require a community research effort.
- **Denys Poshvanyk:** Denys provided critical feedback on the CEL testing principles and the proposed research directions.

Bibliography

- [1] 2018 stack overflow developer survey <https://insights.stackoverflow.com/survey/2018/>.
- [2] 31C3 Schedule Application <https://github.com/tuxmobil/CampFahrplan>.
- [3] 99Tests <https://99tests.com>.
- [4] Adobe Photoshop <http://www.photoshop.com>.
- [5] Airbrake <https://airbrake.io>.
- [6] Android Emulator Documentation <http://developer.android.com/tools/help/emulator.html>.
- [7] Android Fragmentation Statistics <http://opensignal.com/reports/2014/android-fragmentation/>.
- [8] Android InputType Specifications <https://developer.android.com/reference/android/widget/TextView>.
- [9] Android Logcat Debugging Tool <http://developer.android.com/tools/help/logcat.html>.
- [10] Android Platform Install Base Information <https://developer.android.com/about/dashboards/index.html>.
- [11] Android-Studio <https://developer.android.com/studio/index.html>.
- [12] Android UI-Development <https://developer.android.com/guide/topics/ui/overview.html>.
- [13] Android UI/Application Exerciser Monkey <http://developer.android.com/tools/help/monkey.html>.

- [14] Android uiautomator <http://developer.android.com/tools/help/uiautomator/index.html>.
- [15] androidx86 project <http://www.android-x86.org>.
- [16] Androtest Framework <http://bear.cc.gatech.edu/~shauvik/androtest/>.
- [17] Apache Ant Build System <http://ant.apache.org>.
- [18] Apache Cordova <https://cordova.apache.org>.
- [19] Apktool <https://code.google.com/p/android-apktool/>.
- [20] Apperian <https://www.apperian.com>.
- [21] Appetize.io <https://appetize.io>.
- [22] Appium Testing Framework <http://appium.io>.
- [23] Applause <https://www.applause.com>.
- [24] Apple App Store <https://www.apple.com/ios/app-store/>.
- [25] Apple UI-Automation Documentation <https://web.archive.org/web/20140812195854/https://develop>
- [26] AppSee <https://www.appsee.com>.
- [27] Apptimize <https://apptimize.com>.
- [28] Appypie <http://www.appypie.com/app-prototype-builder>.
- [29] AWS Device Farm <https://aws.amazon.com/documentation/devicefarm/>.
- [30] Azetone <https://www.azetone.com>.
- [31] BugClipper <http://bugclipper.com>.
- [32] Calabash Testing Framework <http://calaba.sh>.
- [33] Convolution operator <http://mathworld.wolfram.com/Convolution.html>.

- [34] Crashlytics <https://try.crashlytics.com/>.
- [35] Crashescope online appendix <https://www.android-dev-tools.com/redraw/>.
- [36] CrowdSourcedtesting <https://crowdsourcedtesting.com>.
- [37] CrowdSprint <https://crowdsprint.com>.
- [38] Cucumber <https://cucumber.io>.
- [39] Destructive Software Testing https://en.wikipedia.org/wiki/Destructive_testing.
- [40] Emma Code Coverage Tool <http://emma.sourceforge.net>.
- [41] Espresso Test Recorder <https://developer.android.com/studio/test/espresso-test-recorder.html>.
- [42] Espresso Testing Framework <https://google.github.io/android-testing-support-library/docs/espresso/>.
- [43] Flinto <https://www.flinto.com>.
- [44] Fluid-UI <https://www.fluidui.com>.
- [45] Genymotion emulator <https://www.genymotion.com>.
- [46] Google-api <https://github.com/NeroBurner/googleplay-api>.
- [47] Google Firebase Test Lab <https://firebase.google.com/docs/test-lab/>.
- [48] Google Firebase Test Lab Robo Test <https://firebase.google.com/docs/test-lab/robo-ux-test>.
- [49] Google material design <https://material.io>.
- [50] Google Play Store <https://play.google.com/store?hl=en>.
- [51] Gradle Build System <https://gradle.org>.

- [52] Gvt online appendix <https://www.android-dev-tools.com/gvt/>.
- [53] Intent Fuzzer <https://www.isecpartners.com/tools/mobile-security/intent-fuzzer.aspx>.
- [54] ionic framework <https://ionicframework.com/>.
- [55] Irise <https://www.irise.com/mobile-prototyping/>.
- [56] JustinMind <https://www.justinmind.com>.
- [57] Keynote <http://www.keynote.com>.
- [58] LookBack <https://lookback.io>.
- [59] Loop11 <https://www.loop11.com>.
- [60] The Marketch Plugin for Sketch <https://github.com/tudou527/marketch>.
- [61] MarvelApp <https://marvelapp.com/prototyping/>.
- [62] MatLab Neural Network Toolbox <https://www.mathworks.com/products/neural-network.html>.
- [63] Mobile Apps: What Consumers Really Need and Want https://info.dynatrace.com/rs/compuware/images/Mobile_App_Survey_Report.pdf.
- [64] Mockingbot <https://mockingbot.com>.
- [65] Mockup.io <https://mockup.io/about/>.
- [66] MrTappy <https://www.mrtappy.com>.
- [67] MyCrowdQA <https://mycrowd.com>.
- [68] New Relic. <https://newrelic.com/>.
- [69] OpenCV <https://opencv.org>.

- [70] Pay4Bugs <https://www.pay4bugs.com>.
- [71] Photohawk Library <http://datascience.github.io/photohawk/>.
- [72] Pix2code GitHub Repository <https://github.com/tonybeltramelli/pix2code>.
- [73] Pixate <http://www.pixate.com>.
- [74] ProtoApp <https://prottapp.com/features/>.
- [75] Proto.io <https://proto.io>.
- [76] Qmetry Test Automation Framework <https://qmetry.github.io/qaf/>.
- [77] Quantum <https://community.perfectomobile.com/posts/1286012-introducing-quantum-framework>.
- [78] Ranorex Testing Framework <http://www.ranorex.com>.
- [79] React native <https://facebook.github.io/react-native/>.
- [80] Redraw online appendix <https://www.android-dev-tools.com/redraw/>.
- [81] Remaui web version <http://pixeltoapp.com>.
- [82] Roboelectric Testing Framework <http://roboelectric.org>.
- [83] Robotium Recorder <https://robotium.com/products/robotium-recorder>.
- [84] Robotium Testing Framework <https://robotium.com>.
- [85] SauceLabs <https://saucelabs.com>.
- [86] The Sketch Design Tool <https://www.sketchapp.com>.
- [87] Sketch Extensions <https://www.sketchapp.com/extensions/>.
- [88] Stack-Overflow Q/A site <https://stackoverflow.com>.

- [89] Statista - Mobile Market Share <https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/>.
- [90] STF <https://github.com/openstf/stf>.
- [91] Supernova studio component classification limitation <https://blog.prototypr.io/introducing-supernova-studio-35335de5044c>.
- [92] Supernova studio <https://supernova.studio>.
- [93] SWRVE <https://www.swrve.com>.
- [94] Tesseract OCR Library <https://github.com/tesseract-ocr/tesseract/wiki>.
- [95] Tesseract OCR Library <https://www.mathworks.com/products/computer-vision.html>.
- [96] Tesseract OCR Library <https://www.mathworks.com/products/parallel-computing.html>.
- [97] TestArmy <http://testarmy.com>.
- [98] TestDroid <http://bitbar.com/testing/>.
- [99] TestFairy <https://testfairy.com>.
- [100] TestFlight <https://developer.apple.com/testflight/>.
- [101] TestGrid <https://testgrid.io>.
- [102] Test.io www.test.io.
- [103] Unity Game Engine <https://unity3d.com>.
- [104] UserLytics <https://test.io>.
- [105] UserZoom <http://www.userzoom.com>.

- [106] Vagrant VirtualBox Manager <https://docs.vagrantup.com/v2/>.
- [107] VirtualBox <https://www.virtualbox.org>.
- [108] Visual-Studio <https://www.visualstudio.com>.
- [109] Vysor <http://www.vysor.io>.
- [110] Wagner-Fischer Algorithm https://en.wikipedia.org/wiki/Wagner-Fischer_algorithm.
- [111] WatchSend <https://watchsend.com>.
- [112] Why Your App's UX is More Important than You Think
<http://www.codemag.com/Article/1401041>.
- [113] Xamarin Test Cloud <https://www.xamarin.com>.
- [114] Xamarin Test Recorder <https://www.xamarin.com/test-cloud/recorder>.
- [115] XCode <https://developer.apple.com/xcode/>.
- [116] XDA Developers Forums <https://forum.xda-developers.com>.
- [117] Xiffe <http://xiffe.com>.
- [118] An Automated Oracle for Verifying GUI Objects. *SIGSOFT Softw. Eng. Notes*, 26(4):83–88, July 2001.
- [119] CHRISTOFFER QUIST ADAMSEN, GIANLUCA MEZZETTI, AND ANDERS MØLLER. Systematic Execution of Android Test Suites in Adverse Conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA'15*, pages 83–93, Baltimore, MD, USA, 2015. ACM.
- [120] D. AMALFITANO, A. FASOLINO, P. TRAMONTANA, B. TA, AND A. MEMON. Mo-biGUITAR – A Tool for Automated Model-Based Testing of Mobile Apps. *Software, IEEE*, PP(99):1–1, 2014.

- [121] DOMENICO AMALFITANO, ANNA RITA FASOLINO, PORFIRIO TRAMONTANA, SALVATORE DE CARMINE, AND ATIF M. MEMON. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE'12*, pages 258–261, Essen, Germany, 2012. ACM.
- [122] SASWAT ANAND, MAYUR NAIK, MARY JEAN HARROLD, AND HONGSEOK YANG. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12*, pages 59:1–59:11, Cary, North Carolina, 2012. ACM.
- [123] APPLE. App Store - Support. <https://developer.apple.com/support/app-store/>.
- [124] TANZIRUL AZIM AND IULIAN NEAMTIU. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13*, pages 641–660, Indianapolis, Indiana, USA, 2013. ACM.
- [125] G. BAVOTA, M. LINARES-VÁSQUEZ, C.E. BERNAL-CÁRDENAS, M. DI PENTA, R. OLIVETO, AND D. POSHYVANYK. The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps. *Software Engineering, IEEE Transactions on*, 41(4):384–407, April 2015.
- [126] HERBERT BAY, ANDREAS ESS, TINNE TUYTELAARS, AND LUC VAN GOOL. Speeded-Up Robust Features (SURF). *Comput. Vis. Image Underst.*, 110(3):346–359, June 2008.
- [127] TONY BELTRAMELLI. Pix2code: Generating Code from a Graphical User Interface Screenshot. *CoRR*, abs/1705.07962, 2017.
- [128] NICOLAS BETTENBURG, SASCHA JUST, ADRIAN SCHRÖTER, CATHRIN WEISS, RAHUL PREMRAJ, AND THOMAS ZIMMERMANN. What Makes a Good Bug Report?

- In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 308–318, Atlanta, Georgia, 2008. ACM.
- [129] NICOLAS BETTENBURG, RAHUL PREMRAJ, THOMAS ZIMMERMANN, AND SUNGHUN KIM. Extracting Structural Information from Bug Reports. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 27–30, Leipzig, Germany, 2008. ACM.
- [130] RICHARD BONETT, KAUSHAL KAFLE, KEVIN MORAN, ADWAIT NADKARNI, AND DENYS POSHYVANYK. Discovering flaws in security-focused static analysis tools for android using systematic mutation. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association.
- [131] SILVIA BREU, RAHUL PREMRAJ, JONATHAN SILLITO, AND THOMAS ZIMMERMANN. Information Needs in Bug Reports: Improving Cooperation Between Developers and Users. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, CSCW '10, pages 301–310, Savannah, Georgia, USA, 2010. ACM.
- [132] J. BROOKE. SUS: A quick and dirty usability scale. In *Usability Evaluation in Industry*, P. W. Jordan, B. Weerdmeester, A. Thomas, and I. L. McLelland, editors. Taylor and Francis, London, 1996.
- [133] F. P. J. BROOKS. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.
- [134] ANABELA CAETANO, NERI GOULART, MANUEL FONSECA, AND JOAQUIM JORGE. Javasketchit: Issues in sketching the look of user interfaces. In *AAAI Spring Symposium on Sketch Understanding*, SSS'02, pages 9–14, 2002.
- [135] J. CANNY. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, November 1986.

- [136] RANVEER CHANDRA, BÖRJE F. KARLSSON, NICHOLAS D. LANE, CHIEH-JAN MIKE LIANG, SUMAN NATH, JITU PADHYE, LENIN RAVINDRANATH, AND FENG ZHAO. How to the Smash Next Billion Mobile App Bugs? *GetMobile: Mobile Comp. and Comm.*, 19(1):34–38, June 2015.
- [137] TSUNG-HSIANG CHANG, TOM YEH, AND ROB MILLER. Associating the Visual Representation of User Interfaces with Their Internal Structures and Metadata. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 245–256, Santa Barbara, California, USA, 2011. ACM.
- [138] K. CHARMAZ. *Constructing Grounded Theory*. SAGE Publications Inc., 2006.
- [139] STÉPHANE CHATTY, STÉPHANE SIRE, JEAN-LUC VINOT, PATRICK LECOANET, ALEXANDRE LEMORT, AND CHRISTOPHE MERTZ. Revisiting Visual Interface Programming: Creating GUI Tools for Designers and Programmers. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, UIST '04, pages 267–276, Santa Fe, NM, USA, 2004. ACM.
- [140] KAI CHEN, PENG LIU, AND YINGJUN ZHANG. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 175–186, Hyderabad, India, 2014. ACM.
- [141] WONTAE CHOI, GEORGE NECULA, AND KOUSHIK SEN. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 623–640, Indianapolis, Indiana, USA, 2013. ACM.
- [142] S. R. CHOUDHARY, A. GORLA, AND A. ORSO. Automated Test Input Generation for Android: Are We There Yet? (E). In *2015 30th IEEE/ACM International Confer-*

- ence on Automated Software Engineering (ASE), ASE'15, pages 429–440, November 2015. ISSN:.
- [143] SHAUVIK ROY CHOUDHARY, MUKUL R. PRASAD, AND ALESSANDRO ORSO. Cross-Check: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST '12*, pages 171–180, Washington, DC, USA, 2012. IEEE Computer Society.
- [144] A. CIURUMELEA, A. SCHAUFELBÜHL, S. PANICHELLA, AND H. C. GALL. Analyzing reviews and code of mobile apps for better release planning. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, SANER'17, pages 91–102, February 2017.
- [145] W.J. CONOVER. *Practical Nonparametric Statistics*. Wiley, 1998.
- [146] ADRIEN COYETTE, SUZANNE KIEFFER, AND JEAN VANDERDONCKT. Multi-fidelity Prototyping of User Interfaces. In *Proceedings of the 11th IFIP TC 13 International Conference on Human-Computer Interaction, INTERACT'07*, pages 150–164, Rio de Janeiro, Brazil, 2007. Springer-Verlag.
- [147] CHRISTOPH CSALLNER AND YANNIS SMARAGDAKIS. JCrasher: An Automatic Robustness Tester for Java. *Softw. Pract. Exper.*, 34(11):1025–1050, September 2004.
- [148] CHRISTOPH CSALLNER, YANNIS SMARAGDAKIS, AND TAO XIE. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2):8:1–8:37, May 2008.
- [149] GABRIELLA CSURKA, CHRISTOPHER DANCE, LIXIN FAN, JUTTA WILLAMOWSKI, AND CÉDRIC BRAY. Visual categorization with bags of keypoints. In *Workshop on Statistical Learning in Computer Vision*, volume 1 of *ECCV'04*, pages 1–2. Prague, 2004.

- [150] YINGNONG DANG, RONGXIN WU, HONGYU ZHANG, DONGMEI ZHANG, AND PETER NOBEL. ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity. In *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, pages 1084–1093, Zurich, Switzerland, 2012. IEEE Press.
- [151] BIPLAB DEKA, ZIFENG HUANG, CHAD FRANZEN, JOSHUA HIBSCHMAN, DANIEL AFERGAN, YANG LI, JEFFREY NICHOLS, AND RANJITHA KUMAR. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual Symposium on User Interface Software and Technology, UIST '17*, 2017.
- [152] ANDREA DI SORBO, SEBASTIANO PANICHELLA, CAROL V. ALEXANDRU, JUNJI SHIMAGAKI, CORRADO A. VISAGGIO, GERARDO CANFORA, AND HARALD C. GALL. What Would Users Change in My App? Summarizing App Reviews for Recommending Software Changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE'16*, pages 499–510, Seattle, WA, USA, 2016. ACM.
- [153] MORGAN DIXON AND JAMES FOGARTY. Prefab: Implementing Advanced Behaviors Using Pixel-based Reverse Engineering of Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '10*, pages 1525–1534, Atlanta, Georgia, USA, 2010. ACM.
- [154] MORGAN DIXON, DANIEL LEVENTHAL, AND JAMES FOGARTY. Content and Hierarchy in Pixel-based Methods for Reverse Engineering Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '11*, pages 969–978, Vancouver, BC, Canada, 2011. ACM.
- [155] M. FAZZINI, E. N. D. A. FREITAS, S. R. CHOUDHARY, AND A. ORSO. Barista: A Technique for Recording, Encoding, and Running Platform Independent Android

- Tests. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, ICST'17, pages 149–160, March 2017. ISSN:.
- [156] J. FEIGENSPAN, C. KÄSTNER, J. LIEBIG, S. APEL, AND S. HANENBERG. Measuring Programming Experience. In *ICPC'12*, ICPC'12, pages 73–82, 2012.
- [157] WEI FU AND TIM MENZIES. Easy over Hard: A Case Study on Deep Learning. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, FSE'17*, pages 49–60, Paderborn, Germany, 2017. ACM.
- [158] ROSS GIRSHICK, JEFF DONAHUE, TREVOR DARRELL, AND JITENDRA MALIK. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR '14*, pages 580–587, Washington, DC, USA, 2014. IEEE Computer Society.
- [159] LORENZO GOMEZ, IULIAN NEAMTIU, TANZIRUL AZIM, AND TODD MILLSTEIN. RERAN: Timing- and Touch-sensitive Record and Replay for Android. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE'13*, pages 72–81, San Francisco, CA, USA, 2013. IEEE Press.
- [160] M. GRECHANIK, Q. XIE, AND C. FU. Creating GUI Testing Tools Using Accessibility Technologies. In *2009 International Conference on Software Testing, Verification, and Validation Workshops, ICSTW'09*, pages 243–250, April 2009.
- [161] R. J. GRISSOM AND J. J. KIM. *Effect Sizes for Research: A Broad Practical approach*. Lawrence Earlbaum Associates, 2005.
- [162] ZHONGXIAN GU, E.T. BARR, D.J. HAMILTON, AND ZHENDONG SU. Has the bug really been fixed? In *Software Engineering, 2010 ACM/IEEE 32nd International Conference On*, volume 1 of *ICSE'10*, pages 55–64, May 2010.

- [163] JIAPING GUI, STUART MCILROY, MEIYAPPAN NAGAPPAN, AND WILLIAM G. J. HALFOND. Truth in Advertising: The Hidden Cost of Mobile Ads for Software Developers. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15*, pages 100–110, Florence, Italy, 2015. IEEE Press.
- [164] M. HALPERN, Y. ZHU, R. PERI, AND V. J. REDDI. Mosaic: Cross-platform user-interaction record and replay for the fragmented android ecosystem. In *ISPASS'15*, ISPASS'15, pages 215–224, March 2015.
- [165] D. HAN, C. ZHANG, X. FAN, A. HINDLE, K. WONG, AND W. STROUILA. Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs. In *WCRE'12*, WCRE'12, pages 83–92, 2012.
- [166] SHUAI HAO, BIN LIU, SUMAN NATH, WILLIAM G.J. HALFOND, AND RAMESH GOVINDAN. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '14*, pages 204–217, Bretton Woods, New Hampshire, USA, 2014. ACM.
- [167] K. HE, X. ZHANG, S. REN, AND J. SUN. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, CVPR'16, pages 770–778, June 2016. ISSN:.
- [168] ANNIKA HINZE, JUDY BOWEN, YUTING WANG, AND ROBI MALIK. Model-driven GUI & Interaction Design Using Emulation. In *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '10, pages 273–278, Berlin, Germany, 2010. ACM.
- [169] K. HOLL AND F. ELBERZHAGER. A mobile-specific failure classification and its usage to focus quality assurance. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 385–388, Aug 2014.

- [170] GANG HU, XINHAO YUAN, YANG TANG, AND JUNFENG YANG. Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor. In *Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14*, pages 18:1–18:15, Amsterdam, The Netherlands, 2014. ACM.
- [171] YONGJIAN HU, TANZIRUL AZIM, AND IULIAN NEAMTIU. Versatile Yet Lightweight Record-and-replay for Android. In *OOPSLA'15, OOPSLA 2015*, pages 349–366, Pittsburgh, PA, USA, 2015. ACM.
- [172] A. ISSA, J. SILLITO, AND V. GAROUSI. Visual testing of graphical user interfaces: An exploratory study towards systematic definitions and approaches. In *2012 14th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 11–15, Sept 2012.
- [173] REYHANEH JABBARVAND, ALIREZA SADEGHI, HAMID BAGHERI, AND SAM MALEK. Energy-aware Test-suite Minimization for Android Apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA'16*, pages 425–436, Saarbrücken, Germany, 2016. ACM.
- [174] JOXAN JAFFAR, VIJAYARAGHAVAN MURALI, AND JORGE A. NAVAS. Boosting Concolic Testing via Interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, FSE'13*, pages 48–58, Saint Petersburg, Russia, 2013. ACM.
- [175] CASPER S. JENSEN, MUKUL R. PRASAD, AND ANDERS MØLLER. Automated Testing with Targeted Event Sequence Generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA'13*, pages 67–77, Lugano, Switzerland, 2013. ACM.
- [176] WEI JIN AND ALESSANDRO ORSO. BugRedux: Reproducing Field Failures for In-house Debugging. In *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, pages 474–484, Zurich, Switzerland, 2012. IEEE Press.

- [177] WEI JIN AND ALESSANDRO ORSO. F3: Fault Localization for Field Failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSA'13*, pages 213–223, Lugano, Switzerland, 2013. ACM.
- [178] N. JONES. Seven best practices for optimizing mobile testing efforts. Technical Report G00248240, Gartner.
- [179] M. E. JOORABCHI, M. ALI, AND A. MESBAH. Detecting inconsistencies in multi-platform mobile apps. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, ISSRE'15, pages 450–460, November 2015.
- [180] M.E. JOORABCHI, A. MESBAH, AND P. KRUCHTEN. Real Challenges in Mobile App Development. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium On*, ESEM'12, pages 15–24, October 2013.
- [181] MONA JOORABCHI, MEHDI MIRZAAGHAEI, AND ALI MESBAH. Works for Me! Characterizing Non-reproducible Bug Reports. In *Proceedings of the 11th Working Conference on Mining Software Repositories, MSR'14*, pages 62–71, Hyderabad, India, 2014. ACM.
- [182] UJJWAL KARN. An Intuitive Explanation of Convolutional Neural Nets <https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/>.
- [183] HAMMAD KHALID, EMAD SHIHAB, MEIYAPPAN NAGAPPAN, AND AHMED E. HASSAN. What Do Mobile App Users Complain About? A Study on Free iOS Apps. *IEEE Software*, (2-3):103–134, 2014.
- [184] SUNGHUN KIM, T. ZIMMERMANN, AND N. NAGAPPAN. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference On*, DSN'11, pages 486–493, June 2011.

- [185] P. S. KOCHHAR, F. THUNG, N. NAGAPPAN, T. ZIMMERMANN, AND D. LO. Understanding the Test Automation Culture of App Developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, ICST'15, pages 1–10, April 2015.
- [186] ALEX KRIZHEVSKY, ILYA SUTSKEVER, AND GEOFFREY E HINTON. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, pages 1097–1105. Curran Associates, Inc., 2012.
- [187] K. KUUSINEN AND T. MIKKONEN. Designing User Experience for Mobile Apps: Long-Term Product Owner Perspective. In *2013 20th Asia-Pacific Software Engineering Conference*, volume 1 of *APSEC'13*, pages 535–540, December 2013.
- [188] J. A. LANDAY AND B. A. MYERS. Sketching interfaces: Toward more human interface design. *Computer*, 34(3):56–64, March 2001.
- [189] JAMES A. LANDAY AND BRAD A. MYERS. Interactive Sketching for the Early Stages of User Interface Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 43–50, Denver, Colorado, USA, 1995. ACM Press/Addison-Wesley Publishing Co.
- [190] WALTER S. LASECKI, JUHO KIM, NICK RAFTER, ONKUR SEN, JEFFREY P. BIGHAM, AND MICHAEL S. BERNSTEIN. Apparition: Crowdsourced User Interfaces That Come to Life As You Sketch Them. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 1925–1934, Seoul, Republic of Korea, 2015. ACM.
- [191] Y. LECUN, L. BOTTOU, Y. BENGIO, AND P. HAFFNER. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

- [192] V. LELLI, A. BLOUIN, AND B. BAUDRY. Classifying and qualifying gui defects. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, April 2015.
- [193] V. LELLI, A. BLOUIN, AND B. BAUDRY. Classifying and Qualifying GUI Defects. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, ICST’15, pages 1–10, April 2015.
- [194] CHIEH-JAN MIKE LIANG, NICHOLAS D. LANE, NIELS BROUWERS, LI ZHANG, BÖRJE F. KARLSSON, HAO LIU, YAN LIU, JUN TANG, XIANG SHAN, RANVEER CHANDRA, AND FENG ZHAO. Caiipa: Automated Large-scale Mobile App Testing Through Contextual Fuzzing. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking, MobiCom ’14*, pages 519–530, Maui, Hawaii, USA, 2014. ACM.
- [195] Y. LIN, J. F. ROJAS, E. CHU, AND Y. LAI. On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices. *IEEE Transactions on Software Engineering (TSE)*, (99):1–1, 2014.
- [196] M. LINARES-VÁSQUEZ, C. BERNAL-CARDENAS, K. MORAN, AND D. POSHYVANYK. How do Developers Test Android Applications? In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ICSME’17, pages 613–622, September 2017. ISSN:.
- [197] M. LINARES-VÁSQUEZ, K. MORAN, AND D. POSHYVANYK. Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ICSME’17, pages 399–410, September 2017. ISSN:.
- [198] M. LINARES-VÁSQUEZ, C. VENDOME, Q. LUO, AND D. POSHYVANYK. How developers detect and fix performance bottlenecks in Android apps. In *2015 IEEE Inter-*

- national Conference on Software Maintenance and Evolution (ICSME)*, ICSME'15, pages 352–361, September 2015. ISSN:.
- [199] MARIO LINARES-VÁSQUEZ. Enabling Testing of Android Apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 763–765, Florence, Italy, 2015. IEEE Press.
- [200] MARIO LINARES-VÁSQUEZ, GABRIELE BAVOTA, CARLOS BERNAL-CÁRDENAS, MASSIMILIANO DI PENTA, ROCCO OLIVETO, AND DENYS POSHYVANYK. API Change and Fault Proneness: A Threat to the Success of Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, FSE'13, pages 477–487, Saint Petersburg, Russia, 2013. ACM.
- [201] MARIO LINARES-VÁSQUEZ, GABRIELE BAVOTA, MICHELE TUFANO, KEVIN MORAN, MASSIMILIANO DI PENTA, CHRISTOPHER VENDOME, CARLOS BERNAL-CÁRDENAS, AND DENYS POSHYVANYK. Enabling Mutation Testing for Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, FSE'17, pages 233–244, Paderborn, Germany, 2017. ACM.
- [202] MARIO LINARES-VÁSQUEZ, SAM KLOCK, COLLIN McMILLAN, AMINATA SABANÉ, DENYS POSHYVANYK, AND YANN-GAËL GUÉHÉNEUC. Domain Matters: Bringing Further Evidence of the Relationships Among Anti-patterns, Application Domains, and Quality-related Metrics in Java Mobile Apps. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC'14, pages 232–243, Hyderabad, India, 2014. ACM.
- [203] MARIO LINARES-VÁSQUEZ, MARTIN WHITE, CARLOS BERNAL-CÁRDENAS, KEVIN MORAN, AND DENYS POSHYVANYK. Mining Android App Usages for Generating Actionable GUI-based Execution Scenarios. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 111–122, Florence, Italy, 2015. IEEE Press.

- [204] PENG LIU, JULIAN DOLBY, AND CHARLES ZHANG. Finding Incorrect Compositions of Atomicity. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, FSE'13, pages 158–168, Saint Petersburg, Russia, 2013. ACM.
- [205] GUY LOHMAN, JON CHAMPLIN, AND PETER SOHN. Quickly Finding Known Software Problems via Automated Symptom Matching. In *Proceedings of the Second International Conference on Automatic Computing*, ICAC '05, pages 101–110, Washington, DC, USA, 2005. IEEE Computer Society.
- [206] QI LUO, KEVIN MORAN, AND DENYS POSHYVANYK. A Large-scale Empirical Comparison of Static and Dynamic Test Case Prioritization Techniques. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'16, pages 559–570, Seattle, WA, USA, 2016. ACM.
- [207] QI LUO, KEVIN MORAN, DENYS POSHYVANYK, AND MASSIMILIANO DI PENTA. Assessing test case prioritization on real faults and mutants. In *Proceedings of the 2018 34th International Conference on Software Maintenance and Evolution (ICSME'18)*, ICSME'1816, page to appear, Montpellier, France, 2018. ACM.
- [208] ARAVIND MACHIRY, ROHAN TAHILIANI, AND MAYUR NAIK. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, FSE'13, pages 224–234, Saint Petersburg, Russia, 2013. ACM.
- [209] S. MAHAJAN AND W. G. J. HALFOND. Detection and Localization of HTML Presentation Failures Using Computer Vision-Based Techniques. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, ICST'15, pages 1–10, April 2015.
- [210] S. MAHAJAN, B. LI, P. BEHNAMGHADER, AND W. G. J. HALFOND. Using Visual Symptoms for Debugging Presentation Failures in Web Applications. In *2016 IEEE*

International Conference on Software Testing, Verification and Validation (ICST), ICST'16, pages 191–201, April 2016. ISSN:.

- [211] SONAL MAHAJAN, ABDULMAJEED ALAMEER, PHIL MCMINN, AND WILLIAM G. J. HALFOND. Automated Repair of Layout Cross Browser Issues Using Search-based Techniques. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'17, pages 249–260, Santa Barbara, CA, USA, 2017. ACM.
- [212] RIYADH MAHMOOD, NARIMAN MIRZAEI, AND SAM MALEK. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'14, pages 599–609, Hong Kong, China, 2014. ACM.
- [213] RUPAK MAJUMDAR AND KOUSHIK SEN. Hybrid Concolic Testing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.
- [214] KE MAO, MARK HARMAN, AND YUE JIA. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA'16, pages 94–105, Saarbrücken, Germany, 2016. ACM.
- [215] T. MCDONNELL, B. RAY, AND MIRYUNG KIM. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the 2013 International Conference on Software Maintenance*, ICSM'13, pages 70–79, 2013.
- [216] XIAOJUN MENG, SHENGDONG ZHAO, YONGFENG HUANG, ZHONGYUAN ZHANG, JAMES EAGAN, AND RAMANATHAN SUBRAMANIAN. WADE: Simplified GUI Add-on Development for Third-party Software. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 2221–2230, Toronto, Ontario, Canada, 2014. ACM.

- [217] JAN MESKENS, KRIS LUYTEN, AND KARIN CONINX. Plug-and-design: Embracing Mobile Devices As Part of the Design Environment. In *Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '09, pages 149–154, New York, NY, USA, 2009. ACM.
- [218] NARIMAN MIRZAEI, HAMID BAGHERI, RIYADH MAHMOOD, AND SAM MALEK. SIG-Droid: Automated system input generation for Android applications. In *ISSRE'15*, ISSRE'15, pages 461–471, November 2015.
- [219] K. MORAN, M. LINARES-VÁSQUEZ, C. BERNAL-CÁRDENAS, C. VENDOME, AND D. POSHYVANYK. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, ICST'16, pages 33–44, April 2016. ISSN:.
- [220] K. P. MORAN, C. BERNAL-CÁRDENAS, M. CURCIO, R. BONETT, AND D. POSHYVANYK. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.
- [221] KEVIN MORAN. Enhancing Android Application Bug Reporting. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, FSE'15, pages 1045–1047, Bergamo, Italy, 2015. ACM.
- [222] KEVIN MORAN, CARLOS BERNAL-CÁRDENAS, MARIO LINARES-VÁSQUEZ, AND DENYS POSHYVANYK. Overcoming lanaguge dichotomies: Toward effective program comprehension for mobile app development. In *26th IEEE International Conference on Program Comprehnsion*, May 2018. To Appear.
- [223] KEVIN MORAN, RICHARD BONETT, CARLOS BERNAL-CÁRDENAS, BRENDAN OTTEN, DANIEL PARK, AND DENYS POSHYVANYK. On-Device Bug Reporting for Android Applications. In *MobileSOFT'17*, MobileSoft'17, May 2017.

- [224] KEVIN MORAN, BOYANG LI, CARLOS BERNAL-CÁRDENAS, DAN JELF, AND DENYS POSHYVANYK. Automated Reporting of GUI Design Violations in Mobile Apps. In *Proceedings of the 40th International Conference on Software Engineering Companion*, ICSE '18, page to appear, Gothenburg, Sweden, 2018. IEEE Press.
- [225] KEVIN MORAN, MARIO LINARES-VÁSQUEZ, CARLOS BERNAL-CÁRDENAS, AND DENYS POSHYVANYK. Auto-completing Bug Reports for Android Applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, FSE'15, pages 673–686, Bergamo, Italy, 2015. ACM.
- [226] KEVIN MORAN, MARIO LINARES-VÁSQUEZ, CARLOS BERNAL-CÁRDENAS, AND DENYS POSHYVANYK. FUSION: A Tool for Facilitating and Augmenting Android Bug Reporting. In *ICSE'16*, ICSE'16, May 2016.
- [227] KEVIN MORAN, MARIO LINARES-VÁSQUEZ, CARLOS BERNAL-CÁRDENAS, CHRISTOPHER VENDOME, AND DENYS POSHYVANYK. CrashScope: A Practical Tool for Automated Testing of Android Applications. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 15–18, Buenos Aires, Argentina, 2017. IEEE Press.
- [228] KEVIN MORAN, MICHELE TUFANO, CARLOS BERNAL-CÁRDENAS, MARIO LINARES-VÁSQUEZ, GABRIELE BAVOTA, CHRISTOPHER VENDOME, MASSIMILIANO DI PENTA, AND DENYS POSHYVANYK. Mdroid+: A mutation testing framework for android. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, ICSE '18, pages 33–36, New York, NY, USA, 2018. ACM.
- [229] PETER MORVILLE. *User Experience Design*. http://semanticstudios.com/user_experience_design/.
- [230] B. MYERS, S. Y. PARK, Y. NAKANO, G. MUELLER, AND A. KO. How designers design and program interactive behaviors. In *2008 IEEE Symposium on Visual*

Languages and Human-Centric Computing, VLHCC'08, pages 177–184, September 2008.

- [231] BRAD MYERS. Challenges of HCI Design and Implementation. *Interactions*, 1(1):73–83, January 1994.
- [232] A. T. NGUYEN, T. T. NGUYEN, AND T. N. NGUYEN. Divide-and-Conquer Approach for Multi-phase Statistical Migration for Source Code (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE'15, pages 585–596, November 2015. ISSN:.
- [233] BAO NGUYEN AND ATIF MEMON. An Observe-Model-Exercise* Paradigm to Test Event-Driven Systems with Undetermined Input Spaces. *IEEE Transactions on Software Engineering*, 99(Preprints), 2014.
- [234] BAON. NGUYEN, BRYAN ROBBINS, ISHAN BANERJEE, AND ATIF MEMON. GUI-TAR: An innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, pages 1–41, 2013.
- [235] CARLOS PACHECO AND MICHAEL D. ERNST. Eclat: Automatic Generation and Classification of Test Inputs. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 504–527, Glasgow, UK, 2005. Springer-Verlag.
- [236] CARLOS PACHECO, SHUVENDU K. LAHIRI, MICHAEL D. ERNST, AND THOMAS BALL. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.
- [237] F. PALOMBA, M. LINARES-VÁSQUEZ, G. BAVOTA, R. OLIVETO, M. DI PENTA, D. POSHYVANYK, AND A. DE LUCIA. User reviews matter! Tracking crowdsourced

- reviews to support evolution of successful apps. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ICSME'15, pages 291–300, September 2015. ISSN:.
- [238] FABIO PALOMBA, PASQUALE SALZA, ADELINA CIURUMELEA, SEBASTIANO PANICHELLA, HARALD GALL, FILOMENA FERRUCCI, AND ANDREA DE LUCIA. Recommending and Localizing Change Requests for Mobile Apps Based on User Reviews. In *Proceedings of the 39th International Conference on Software Engineering, ICSE '17*, pages 106–117, Piscataway, NJ, USA, 2017. IEEE Press.
- [239] LENIN RAVINDRANATH, SUMAN NATH, JITENDRA PADHYE, AND HARI BALAKRISHNAN. Automatic and Scalable Fault Detection for Mobile Applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '14*, pages 190–203, Bretton Woods, New Hampshire, USA, 2014. ACM.
- [240] SHAOQING REN, KAIMING HE, ROSS GIRSHICK, AND JIAN SUN. Faster R-CNN: Towards Real-time Object Detection with Region Proposal Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems, NIPS'15*, pages 91–99, Montreal, Canada, 2015. MIT Press.
- [241] ROVO89. Xposed Module Repository <http://repo.xposed.info/>.
- [242] SHAUVIK ROY CHOUDHARY, MUKUL R. PRASAD, AND ALESSANDRO ORSO. XPERT: Accurate Identification of Cross-browser Issues in Web Applications. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 702–711, San Francisco, CA, USA, 2013. IEEE Press.
- [243] SHAUVIK ROY CHOUDHARY, MUKUL R. PRASAD, AND ALESSANDRO ORSO. Cross-platform Feature Matching for Web Applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA'14*, pages 82–92, San Jose, CA, USA, 2014. ACM.

- [244] SHAUVIK ROY CHOUDHARY, HUSAYN VERSEE, AND ALESSANDRO ORSO. WEB-DIFF: Automated Identification of Cross-browser Issues in Web Applications. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance, ICSM '10*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.
- [245] ISRAEL MOJICA RUIZ, MEIYAPPAN NAGAPPAN, BRAM ADAMS, THORSTEN BERGER, STEFFEN DIENST, AND AHMED HASSAN. On the relationship between the number of ad libraries in an android app and its rating. *IEEE Software*, (1):1–1, 2014.
- [246] OLGA RUSSAKOVSKY, JIA DENG, HAO SU, JONATHAN KRAUSE, SANJEEV SATHEESH, SEAN MA, ZHIHENG HUANG, ANDREJ KARPATHY, ADITYA KHOSLA, MICHAEL BERNSTEIN, ALEXANDER C. BERG, AND LI FEI-FEI. ImageNet Large Scale Visual Recognition Challenge. *Int. J. Comput. Vision*, 115(3):211–252, December 2015.
- [247] IFLAAH SALMAN, AYSE TOSUN MISIRLI, AND NATALIA JURISTO. Are Students Representatives of Professionals in Software Engineering Experiments? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15*, pages 666–676, Florence, Italy, 2015. IEEE Press.
- [248] H. SAMIR AND A. KAMEL. Automated reverse engineering of Java graphical user interfaces for web migration. In *2007 ITI 5th International Conference on Information and Communications Technology, ICICT'07*, pages 157–162, December 2007.
- [249] RAIMONDAS SASNAUSKAS AND JOHN REGEHR. Intent Fuzzer: Crafting Intentions of Death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis and Software and System Performance Testing, Debugging, and Analytics, WODA+PERTEA'14*, pages 1–5, San Jose, CA, USA, 2014. ACM.
- [250] JULIAN SEIFERT, BASTIAN PFLEGING, ELBA DEL CARMEN VALDERRAMA BAHAMÓNDEZ, MARTIN HERMES, ENRICO RUKZIO, AND ALBRECHT SCHMIDT. Mo-

- bidev: A Tool for Creating Apps on Mobile Phones. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI'11)*, MobileHCI '11, pages 109–112, Stockholm, Sweden, 2011. ACM.
- [251] HYUNMIN SEO AND SUNGHUN KIM. Predicting Recurring Crash Stacks. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012*, pages 180–189, Essen, Germany, 2012. ACM.
- [252] HYUNMIN SEO AND SUNGHUN KIM. How We Get There: A Context-guided Search Strategy in Concolic Testing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE'14*, pages 413–424, Hong Kong, China, 2014. ACM.
- [253] EESHAN SHAH AND ELI TILEVICH. Reverse-engineering User Interfaces to Facilitate porting to and Across Mobile Devices and Platforms. In *Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPEs'11, NEAT'11, & VMIL'11, SPLASH '11 Workshops*, pages 255–260, Portland, Oregon, USA, 2011. ACM.
- [254] TIAGO SILVA DA SILVA, ANGELA MARTIN, FRANK MAURER, AND MILENE SILVEIRA. User-Centered Design and Agile Methods: A Systematic Review. In *Proceedings of the 2011 Agile Conference, AGILE '11*, pages 77–86, Washington, DC, USA, 2011. IEEE Computer Society.
- [255] KAREN SIMONYAN AND ANDREW ZISSERMAN. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, abs/1409.1556, 2014.
- [256] KLAAS-JAN STOL, PAUL RALPH, AND BRIAN FITZGERALD. Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, pages 120–131, New York, NY, USA, 2016. ACM.

- [257] CHRISTIAN SZEGEDY, WEI LIU, YANGQING JIA, PIERRE SERMANET, SCOTT REED, DRAGOMIR ANGUELOV, DUMITRU ERHAN, VINCENT VANHOUCHE, AND ANDREW RABINOVICH. Going Deeper with Convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, CVPR'15, 2015.
- [258] JULIAN THOMÉ, ALESSANDRA GORLA, AND ANDREAS ZELLER. Search-based Security Testing of Web Applications. In *Proceedings of the 7th International Workshop on Search-Based Software Testing*, SBST'14, pages 5–14, Hyderabad, India, 2014. ACM.
- [259] ALLEN B. TUCKER. *Computer Science Handbook, Second Edition*. Chapman & Hall/CRC, 2004.
- [260] HEILA VAN DER MERWE, BRINK VAN DER MERWE, AND WILLEM VISSER. Execution and Property Specifications for JPF-android. *SIGSOFT Softw. Eng. Notes*, 39(1):1–5, February 2014.
- [261] LORENZO VILLARROEL, GABRIELE BAVOTA, BARBARA RUSSO, ROCCO OLIVETO, AND MASSIMILIANO DI PENTA. Release Planning of Mobile Apps Based on User Reviews. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 14–24, New York, NY, USA, 2016. ACM.
- [262] K. WALLACE, K. MORAN, E. NOVAK, G. ZHOU, AND K. SUN. Toward sensor-based random number generation for mobile and iot devices. *IEEE Internet of Things Journal*, 3(6):1189–1201, Dec 2016.
- [263] ZHENHUA WANG, XINGXING WANG, AND GANG WANG. Learning Fine-grained Features via a CNN Tree for Large-scale Classification. *CoRR*, abs/1511.04534, 2015.
- [264] L. WEI, Y. LIU, AND S. C. CHEUNG. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *2016 31st IEEE/ACM*

International Conference on Automated Software Engineering (ASE), ASE'16, pages 226–237, September 2016. ISSN:.

- [265] M. WHITE, M. LINARES-VÁSQUEZ, P. JOHNSON, C. BERNAL-CÁRDENAS, AND D. POSHYVANYK. Generating Reproducible and Replayable Bug Reports from Android Application Crashes. In *2015 IEEE 23rd International Conference on Program Comprehension*, ICPC'15, pages 48–59, May 2015.
- [266] RONGXIN WU, HONGYU ZHANG, SHING-CHI CHEUNG, AND SUNGHUN KIM. CrashLocator: Locating Crashing Faults Based on Crash Stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA'14, pages 204–214, San Jose, CA, USA, 2014. ACM.
- [267] Q. XIE, M. GRECHANIK, C. FU, AND C. CUMBY. Guide: A GUI differentiator. In *2009 IEEE International Conference on Software Maintenance*, ICSM'09, pages 395–396, September 2009.
- [268] WEI YANG, MUKUL R. PRASAD, AND TAO XIE. A Grey-box Approach for Automated GUI-model Generation of Mobile Applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 250–265, Rome, Italy, 2013. Springer-Verlag.
- [269] HUI YE, SHAOYIN CHENG, LANBO ZHANG, AND FAN JIANG. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, MoMM '13, pages 68:68–68:74, Vienna, Austria, 2013. ACM.
- [270] HECTOR YEE, SUMANITA PATTANAIK, AND DONALD P. GREENBERG. Spatiotemporal Sensitivity and Visual Attention for Efficient Rendering of Dynamic Environments. *ACM Trans. Graph.*, 20(1):39–65, January 2001.

- [271] CHAO CHUN YEH, HAN LIN LU, CHUN YEN CHEN, KEE KIAT KHOR, AND SHIH KUN HUANG. CRAXDroid: Automatic Android System Testing by Selective Symbolic Execution. In *Proceedings of the 2014 IEEE Eighth International Conference on Software Security and Reliability-Companion, SERE-C '14*, pages 140–148, Washington, DC, USA, 2014. IEEE Computer Society.
- [272] SHIN YOO, MARK HARMAN, AND DAVID CLARK. Fault Localization Prioritization: Comparing Information-theoretic and Coverage-based Approaches. *ACM Trans. Softw. Eng. Methodol.*, 22(3):19:1–19:29, July 2013.
- [273] RAZIEH NOKHBEH ZAEEM, MUKUL R. PRASAD, AND SARFRAZ KHURSHID. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation, ICST '14*, pages 183–192, Washington, DC, USA, 2014. IEEE Computer Society.
- [274] CLEMENS ZEIDLER, CHRISTOF LUTTEROTH, WOLFGANG STUERZLINGER, AND GERALD WEBER. Evaluating Direct Manipulation Operations for Constraint-Based Layout. In *Human-Computer Interaction – INTERACT 2013: 14th IFIP TC 13 International Conference, Cape Town, South Africa, September 2-6, 2013, Proceedings, Part II*, Paula Kotzé, Gary Marsden, Gitte Lindgaard, Janet Wesson, and Marco Winckler, editors, pages 513–529. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
- [275] MATTHEW D. ZEILER AND ROB FERGUS. Visualizing and Understanding Convolutional Networks. In *Computer Vision – ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I*, David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, pages 818–833. Springer International Publishing, Cham, 2014.

- [276] ANDREAS ZELLER. Isolating Cause-effect Chains from Computer Programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 1–10, Charleston, South Carolina, USA, 2002. ACM.
- [277] ANDREAS ZELLER. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
- [278] HAILONG ZHANG AND ATANAS ROUNTEV. Analysis and Testing of Notifications in Android Wear Applications. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 347–357, Piscataway, NJ, USA, 2017. IEEE Press.
- [279] JIAN ZHOU, HONGYU ZHANG, AND DAVID LO. Where Should the Bugs Be Fixed? - More Accurate Information Retrieval-based Bug Localization Based on Bug Reports. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 14–24, Zurich, Switzerland, 2012. IEEE Press.