3rd International Workshop on Software Development Lifecycle for Mobile (DeMobile 2015)

Proceedings

Aharon Abadi, Shah Rukh Humayoun, and Henry Muccini

August 31, 2015
Bergamo, Italy

The Association for Computing Machinery, Inc.
2 Penn Plaza, Suite 701
New York, NY 10121-0701

ACM ISBN: 978-1-4503-3815-8

Additional copies may be ordered prepaid from:

	Phone: 1-800-342-6626
ACM Order Department	(U.S.A. and Canada)
P.O. BOX 11405	+1-212-626-0500
Church Street Station	(All other countries)
New York, NY 10286-1405	Fax: +1-212-944-1318
	E-mail: acmhelp@acm.org

Production: Conference Publishing Consulting
D-94034 Passau, Germany, info@conference-publishing.com

Foreword

We would like to welcome you to the 3rd International Workshop on Software Development Lifecycle for Mobile (DeMobile 2015), where we aim at establishing a community of researchers and practitioners to share their work and lead further research in the mobile software engineering.

Mobile application usage and development is experiencing exponential growth. The current mobile domain presents new challenges to software engineering. Mobile platforms are rapidly changing, including diverse capabilities as GPS, sensors, and input modes. Activated on mobile platforms, modern applications must be elastic and scale on demand according to the hardware abilities. Applications often need to support and use third-party services. Therefore, during development, security and authorization processes for the dataflow must be applied. Developing such applications requires suitable practices and tools, e.g., architecture techniques that relate to the complexity at hand; improved refactoring tools for hybrid applications using dynamic languages and polyglot development and applications; and testing techniques for applications that run on different devices. Targeting these concerns, the workshop is dedicated to achieve several goals, e.g.: to develop relationships to create a vibrant research community in the area of mobile software development, and to identify the most important research problems for mobile software development.

The first two versions of the workshop were held in conjunction with the 21st and 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2013 & FSE 2014) in Saint Petersburg, Russia and Hong Kong, China respectively. This year, we are delighted to conduct it in conjunction with the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015) in Bergamo, Italy on August 31, 2015.

Researchers and practitioners were invited to submit contributions including research papers of 8 pages long, emerging ideas of 4 pages long, and in-practice experience of 2 pages long extended abstract. Each submission was reviewed by at least three program committee (PC) members, which led to a total number of four accepted papers (2 long and 2 short papers). The acceptance rate for research papers in long and emerging ideas was 50%. In addition, we have 2 distinguished keynote speakers, Prof. Mark Harman from academia and Dr. Yael Dubinsky from industry. Further, we have 7 invited talks with a mixture of academia, research institutes and industry; who will share their research and experiences in the different areas of mobile software engineering. We are grateful for the time and effort the PC members spent in the selection process. Your attendance in the workshop will provide the opportunity for joint discussions about the solved and unsolved problems in mobile software engineering.

Aharon Abadi, Shah Rukh Humayoun, and Henry Muccini
DeMobile 2015 Organizers

DeMobile 2015 Organization

Organizing Committee

Aharon Abadi	IBM Research – Haifa, Israel
Shah Rukh Humayoun	University of Kaiserslautern, Germany
Henry Muccini	University of L'Aquila, Italy

Program Committee:

Matthias Book	University of Iceland, Iceland
Yael Dubinsky	IBM Haifa Research Lab, Israel
Yishai A. Feldman	IBM Research – Haifa, Israel
Lori Flynn	Carnegie Mellon University, USA
Tiziana Catarci	Sapienza University of Rome, Italy
Vincenzo Grassi	University of Roma "Tor Vergata", Italy
Jeff Gray	University of Alabama, USA
Mark Harman	University College London, United Kingdom
Grace Lewis	Carnegie Mellon Software Engineering Institute, USA
Seng Loke	La Trobe University, Australia
Ivano Malavolta	Gran Sasso Science Institute, Italy
Sam Malek	George Mason University, USA
Shahar Maoz	Tel Aviv University, Israel
Vinayak Naik	IIIT-Delhi, India
Marco Pistoia	IBM T. J. Watson Research Center, USA
Rafael Prikladnicki	PUCRS, Brazil
Antonino Sabetta	SAP Research Sophia-Antipolis, France
Federica Sarro	University College London, United Kingdom
Jocelyn Simmonds	Universidad de Chile, Chile
Alin Stefanescu	University of Bucharest, Romania
Shingo Takada	Keio University, Japan
Shmuel Tyszberowicz	The Academic College of Tel-Aviv Yaffo, Israel
Amiram Yehudai	Tel Aviv University, Israel

Contents

App Store Mining and Analysis (Keynote)

Afnan Al-Subaihin, Anthony Finkelstein, Mark Harman*, Yue Jia,
William Martin, Federica Sarro and Yuanyuan Zhang
Department of Computer Science, University College London, London, UK

ABSTRACT

App stores are not merely disrupting traditional software deployment practice, but also offer considerable potential benefit to scientific research. Software engineering researchers have never had available, a more rich, wide and varied source of information about software products. There is some source code availability, supporting scientific investigation as it does with more traditional open source systems. However, what is important and different about app stores, is the other data available. Researchers can access user perceptions, expressed in rating and review data. Information is also available on app popularity (typically expressed as the number or rank of downloads). For more traditional applications, this data would simply be too commercially sensitive for public release. Pricing information is also partially available, though at the time of writing, this is sadly submerging beneath a more opaque layer of in-app purchasing. This talk will review research trends in the nascent field of App Store Analysis, presenting results from the UCL app Analysis Group (UCLappA) and others, and will give some directions for future work.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.8 [**Software Engineering**]: Metrics

General Terms

Design, Experimentation, Measurement

Keywords

App stores, Mining Software Repositories

1. APP STORE MINING AND ANALYSIS

We believe that app stores are scientifically, technically, sociologically and commercially very different from traditional software deployment mechanisms [7, 19, 20]. In particular, they create a software ecosystem [14] that provides researchers with exciting opportunities, not previously available for software engineering research.

In 2012, we set out a research agenda for App Store Mining and Analysis, motivated as follows:

> "*never before has there been a nexus of readily available information that combines the users' view, the developers' claims and the sales information pertinent to a large corpus of software products from many different providers. The combination of these three types of information provides a rich and inter-related set of data from which we can analyse and understand this new software engineering paradigm of app development.*" [7]

This keynote, will review our progress and future directions in the development of this research agenda. It will discuss the importance of features [7, 10] as a suitable level of abstraction with which to discuss apps and app stores, presenting initial results about the migration of features through app stores [24]. The keynote will also consider the ways in which genetic improvement [8,13], can be used to improve existing software systems semi-automatically. We will focus on possibilities for improving energy consumption [1], and dynamic adaptivity, which we believe could be applied to Mobile devices [6] and the management and extension of their product lines [5].

Our group is one of many working on App Store Mining and Analysis. The keynote will also attempt to cover some of the exciting work by other researchers on App Store mining and analysis.

Unlike traditional software deployment mechanisms, we have available, in the App Store, considerable information in the form of customer feedback. This has allowed a great deal of App Store Analysis that investigates this feedback [2, 4, 9, 11, 12, 17, 21, 23, 26]. The keynote will also discuss some of the issues raised by the inherent sampling bias in such empirical studies of app stores [18].

There is also considerable potential in the analysis of the source code [16], requested permissions [22], and API calls [3] of the apps themselves, which is enriched by the contextual information from the App Stores in which they reside. Gorla et al. [3] explore API calls as a cheap and effective proxy for apps' semantic behaviour, while Linares-Vasquez et al. [16] study clones in Android apps, and maintenance [15]. Syer et al. [25] investigate the platform dependence of app defects.

We hope that this keynote will serve to stimulate further interest in the App Store Ecosystems, their mining and analysis and the new software engineering challenges and opportunities they create.

*This keynote will be given by Mark Harman, but reports the joint work of the UCLappA group: A. Al-Subaihin, A. Finkelstein, Y. Jia, W. Martin, F. Sarro, and Y. Zhang. UCLappA website: `http://www0.cs.ucl.ac.uk/staff/F.Sarro/projects/UCLappA/home.html`

DeMobile'15, August 31, 2015, Bergamo, Italy
ACM. 978-1-4503-3815-8/15/08
http://dx.doi.org/10.1145/2804345.2804346

2. REFERENCES

[1] B. Bruce, J. Petke, and M. Harman. Reducing energy consumption using genetic improvement. In *Genetic and evolutionary computation conference (GECCO 2015)*, Madrid, Spain, July 2015.

[2] B. Fu, J. Lin, L. Li, C. Faloutsos, J. Hong, and N. Sadeh. Why people hate your app: Making sense of user feedback in a mobile app store. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1276–1284. ACM, 2013.

[3] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *36th International Conference on Software Engineering (ICSE 2014)*, pages 1025–1035, 2014.

[4] E. Guzman and W. Maalej. How do users like this feature? a fine grained sentiment analysis of app reviews. In *Requirements Engineering (RE 2014)*, pages 153–162, Aug 2014.

[5] M. Harman, Y. Jia, J. Krinke, B. Langdon, J. Petke, and Y. Zhang. Search based software engineering for software product line engineering: a survey and directions for future work (keynote paper). In 18th *International Software Product Line Conference (SPLC 14)*, pages 5–18, Florence, Italy, September 2014.

[6] M. Harman, Y. Jia, W. B. Langdon, J. Petke, I. H. Moghadam, S. Yoo, and F. Wu. Genetic improvement for adaptive software engineering (keynote). In 9th *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2014)*, pages 1–4, New York, NY, USA, 2014. ACM.

[7] M. Harman, Y. Jia, and Y. Zhang. App Store Mining and Analysis: MSR for App Stores. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR '12)*, pages 108–111, Zurich, Swiss, June 2012. IEEE.

[8] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The GISMOE challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In 27th *IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, pages 1–14, Essen, Germany, September 2012.

[9] L. Hoon, R. Vasa, J.-G. Schneider, and J. Grundy. An analysis of the mobile app review landscape: Trends and implications, 2014. available on line from Swinbourne University of Tethnology, Australia.

[10] C. Iacob and R. Harrison. Retrieving and Analyzing Mobile App Feature Requests from Online Reviews. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*, San Francisco, California, USA, 18-19 May 2013.

[11] H. Khalid. On identifying user complaints of iOS apps. In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *35th International Conference on Software Engineering (ICSE 2013)*, pages 1474–1476. IEEE/ACM, 2013.

[12] H. Khalid, E. Shihab, M. Nagappan, and A. Hassan. What do mobile app users complain about? A study on free iOS apps. *IEEE Software*, 32(3):70–77, 2014.

[13] W. B. Langdon and M. Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation (TEVC)*, 2014. To appear.

[14] S. L. Lim and P. J. Bentley. Investigating app store ranking algorithms using a simulation of mobile app ecosystems. In *IEEE Congress on Evolutionary Computation*, pages 2672–2679, 2013.

[15] M. Linares-Vásquez. Supporting evolution and maintenance of android apps. In *36th International Conference on Software Engineering (ICSE 2014) Doctoral Symposium*, pages 714–717, 2014.

[16] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. In *11th Working Conference on Mining Software Repositories (MSR 2014)*, pages 242–251, 2014.

[17] W. Maalej and H. Nabil. Bug report, feature request, or simply praise? on automatically classifying app reviews. In *Requirements Engineering (RE '15)*, 2015. to appear.

[18] W. Martin, M. Harman, Y. Jia, F. Sarro, and Y. Zhang. The app sampling problem for app store mining. In *Mining Software Repositories (MSR'15)*, Florence, Italy, May 2015.

[19] T. Menzies. Beyond data mining; towards "Idea Engineering". In 9th *International Conference on Predictive Models in Software Engineering, PROMISE '13*, Baltimore, MD, USA, Oct. 2013. ACM.

[20] R. Minelli and M. Lanza. Software Analytics for Mobile Applications - Insights & Lessons Learned. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR '13)*, Genova, Italy, 5-8 March 2013. IEEE.

[21] D. Pagano and W. Maalej. User feedback in the appstore: An empirical study. In *requirements engineering (RE 2013)*, pages 125–134. IEEE, 2013.

[22] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the 22nd USENIX Security Symposium*, Washington DC, USA, 14-16 August 2013.

[23] I. J. M. Ruiz, M. Nagappan, A. Bra, T. Berger, S. Dienst, and A. E. Hassan. On the relationship between the number of ad libraries in an android app and its rating, 2014. available on line from Queen's University, Canada.

[24] F. Sarro, A. AlSubaihin, M. Harman, Y. Jia, W. Martin, and Y. Zhang. Feature lifecycles as they spread, migrate, remain and die in app stores. In *Requirements Engineering (RE'15)*, Ottawa, Canada, August 2015. To appear.

[25] M. D. Syer, M. Nagappan, B. Adams, and A. E. Hassan. Studying the relationship between source code quality and mobile platform dependence. *Software Quality Journal*, 2014. To appear; available online.

[26] S. E. S. Taba, I. Keivanloo, Y. Zou, J. Ng, and T. Ng. An exploratory study on the relation between user interface complexity and the perceived quality of android applications. In *International Conference on Web Engineering (ICWE 2014)*, 2014. Late Breaking Result.

Walking the Model: The Smart Mobile Field Engineer (Keynote)

Yael Dubinsky
IBM Research – Haifa, Israel
dubinsky@il.ibm.com

ABSTRACT

Walking the Model (WtM) is a concept that aims to promote the practice of practitioners and crews by providing on-the-job interaction between the system model of the business and the physical world. Among various implementations of this idea, WtM provides mobile field engineers with the ability to view, update and simulate by asking what-if questions while in the field. In this keynote speech, I present the notion and features of walking the model and show how WtM changes the way we perceive of different practices for both actual practice and learning processes.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: User interfaces; H.5.3 [Group and Organization Interfaces]: Theory and models.

General Terms

Performance, Design, Human Factors.

Keywords

Walking the model, on-the-job simulation-based interaction.

1. WALKING THE MODEL CONCEPT

Walking the Model is a concept that aims to promote the practice of practitioners and crews by providing on-the-job interaction between the system model of the business and the physical world.

Any industry concerning field service employs a business infrastructure that is composed of some combination of business assets, machinery (e.g., sensors, actuators), data, and the relationship among them. Such infrastructure can be described at different levels of abstraction – what we call the *System Model,* or shortly, the Model. For example, in Energy companies, the power grid may be referred to as a core part of the model; or in Smart Cities, the public illumination grid may be referred to as model part, etc. Regardless of the particular domain, the model itself is usually maintained using some computerized means that are in the veins of the business operations. Hence, having a faithful representation of the actual situations in the real world at all times, as depicted by the model, is critical to operational efficiency.

DeMobile'15, August 31, 2015, Bergamo, Italy
ACM. 978-1-4503-3815-8/15/08
http://dx.doi.org/10.1145/2804345.2804347

Walking-The-Model allows putting the relevant model in the center, aiming to cut losses by facilitating more efficient maintenance of the model. The current reality is that it is cumbersome to keep organizational information systems up to date in the corresponding operational systems. Our approach focusing on the interaction with the model implies that various operations performed by independent practitioners are all synchronized with the model as the central point of truth. Thus, model accuracy is kept at all times, and indirectly, the model itself becomes centric to the synchronization of all other business operations. Using the model as the baseline, all other aspects pertaining to the business supply chain and operations can be put in concert with it e.g., work order prioritization, asset management, etc.

WtM includes four fundamental interaction operation types that can be performed by crewmembers as well as by the system operations:

The ***View*** category of operations allows practitioners to inquire about the characteristics of a certain model component in real-time while engaging with the corresponding model component in the physical world.

The ***Record*** category of operations enables practitioners to document the knowledge gathered along physical encounters with a certain model component, while in the field in a timely manner. Such knowledge is gathered by associating model components with various means such as photos, textual annotations, and note scans.

The ***Validate*** category of operations provides practitioners with the ability to test different "what-if" scenarios that refer to a certain physical component, and examine, on the model itself, the possible outcomes of the alternative actions prior to their actual pursuing.

The ***Update*** category of operations allows practitioners to modify existing knowledge properties already recorded about model components in the system based on encounters with these components in the physical world, while conforming to the approval and related governance mechanisms associated with such modifications.

2. VALIDATING THE WTM CONCEPT

With an aim to validate the WtM concept, we developed a mobile application to support the synchronization of field engineers' actions in the Energy field service. The app provides the use of the mobile device location service (e.g., GPS), and QR-code markings of the various components in the physical world.

2.1 Accuracy of the Energy Distribution Grid

Within the domain of Energy and Utilities, as it may seem almost obvious, quality of service is a key to the success of the domain suppliers. Maintaining consistent and valid view of operations is crucial both for business efficiency and for end user satisfaction. One such view is the Connectivity model, which is a model representing the various electricity power sources, distributors, transformers, consumers, and their inter-connecting power lines, all comprising the electricity grid. The Connectivity model is the main working tool according to which all maintenance and restoration activities are derived. Currently there is insufficient availability of technical means that would facilitate and ensure the faithfulness and timely representation of the real system status, potentially leading to excessive downtime and monetary losses.

WtM application aims at mitigating the burden of consciously maintaining model accuracy, bridging between the physical and the virtual model views via a designated mobile application that is put in the hands of field engineers to seamlessly keep track of their day-to-day actions. Granting the ability for the entire ecosystem to work with a reliable connectivity model can dramatically boost the efficiency of the entire value chain.

Following is one of the representative use cases that were developed based on close familiarity with processes in Energy services.

2.2 Field Inspection and Work Initiation

This use case involves field crews performing actual field inspection work. This work is typically planned and requires a crew to visit a specific set of components on the network in order to validate the components, record specific test results as requested, validate information already known about the components and provide any further observations. Today this is performed with limited context of the component in its relationship to the power network, its geospatial relationship and any in-field or back-office real-time analytics. Table 1 describes the details including WtM synthesis that is marked in each step.

2.3 Using the WtM Application

The WtM application aims to support the fundamental interaction operations for the field engineers to perform their work. Figure 1 shows the simulation view of the WtM mobile app in which the field crew can ask what-if question whether replacing the malfunctioned transformer with another one provides a valid model. This should be checked against domain specific analytics.

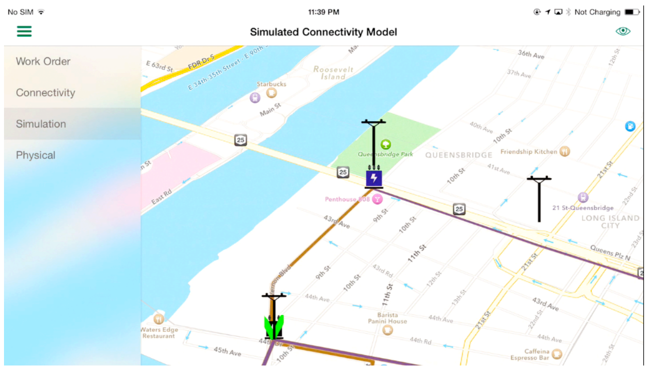


Figure 1: WtM simulation view

Table 1: Field inspection using WtM

Actor Name	*Description of Step*
Field Crew	The crew starts their workday. WtM automatically downloads all their planned work and present a geospatial view of the locations of the work, the power network and its current known state, locations of emergency services and locations of other crews. The driving directions from their current location to the first work location are shown. [VIEW]
SYSTEM	The field crew arrives at the first location. The device automatically indicates the location. If in connected mode, WtM constantly provides location information back to host systems for operations. [UPDATE]
Field Crew	The field crew acknowledges that they are going to proceed with their work. [UPDATE]
SYSTEM	The field crew is going to perform a pole inspection. All the information regarding the pole such as type (type, date installed, prior inspection details, etc.) are provided to the crew in both textual dialogs in a spatial view. The data from relevant sensors e.g. real time reading from devices such as transformers that are attached to the pole, is shown too. [UPDATE]
Field Crew	The crew repositions the location of the pole based on the current location of the mobile device. This immediately changes the location of the pole in the model view. [UPDATE]
Field Crew	The crew notes that the pole is encumbered by vegetation; they take a photo of the pole and the information is saved using WtM. [RECORD]
Field Crew	The crew performs a pole test and enters results into the appropriate work order. [UPDATE]
Advanced Analytics	Based on the result of the pole test, WtM app performs some analytics based on prior tests and current test results and flags to the crew member that the pole requires immediate replacement. [VALIDATE]
Field Crew	The crew acknowledges the pole defect, signals that the work must be scheduled. [UPDATE]
SYSTEM	WtM app completes the work, sends updates and work request to back end systems, shows the pole status of replacement pending. [VIEW]

2.4 Learning Processes

The WtM concept and mobile app enhance field engineering by providing on-the-job mechanisms for learning and improving the practice. Specifically, using simulation to ask what-if questions thus consulting with a simulator before performing a certain action while in the field, provides on-line learning and increases the confidence and professionalism of practitioners.

3. ACKNOWLEDGMENTS

This keynote speech is based on a joint work with lior Limonad, Henry Broodney, Uri Shani, Boris Daitch, and Peter Ruppert at IBM.

AGRippin: A Novel Search Based Testing Technique for Android Applications

Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana
Department of Electrical Engineering and Information Technologies
University Federico II of Naples
Via Claudio 21, Naples, Italy
{domenico.amalfitano, nicola.amatucci, anna.fasolino, porfirio.tramontana}@unina.it

ABSTRACT

Recent studies have shown a remarkable need for testing automation techniques in the context of mobile applications. The main contributions in literature in the field of testing automation regard techniques such as Capture/Replay, Model Based, Model Learning and Random techniques. Unfortunately, only the last two typologies of techniques are applicable if no previous knowledge about the application under testing is available. Random techniques are able to generate effective test suites (in terms of source code coverage) but they need a remarkable effort in terms of machine time and the tests they generate are quite inefficient due to their redundancy. Model Learning techniques generate more efficient test suites but often they do not not reach good levels of coverage. In order to generate test suites that are both effective and efficient, we propose in this paper AGRippin, a novel Search Based Testing technique founded on the combination of genetic and hill climbing techniques. We carried out a case study involving five open source Android applications that has demonstrated how the proposed technique is able to generate test suites that are more effective and efficient than the ones generated by a Model Learning technique.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Experimentation

Keywords

Android, Search Based Testing, Genetic Algorithms

1. INTRODUCTION

The diffusion of smartphones and other mobile devices has grown exponentially in the last years with a corresponding growth of the number and of the complexity of the developed applications. These applications are often realized in a very rapid way, with high pressure due to a very small time-to-market. Moreover, most of the applications are developed by small teams or single developers, that devoted very limited time and resources to design and testing activities. A recent study of Kochhar et al. [14], based on open source Android projects hosted on GitHub, has shown that practices related to testing automation are rarely diffused. In particular, they found that only 14% of the apps they have analyzed contains executable test cases and only 4 apps out of 627 have test cases able to cover more than 40% of the source code of the applications. More generally, Muccini et al. [20] have stated the need for testing automation techniques applicable to the context of mobile applications. Joorabchi et al. [13] on the basis of interviews to 12 senior mobile app developers have concluded that the practice of manual testing is nowadays prevalent for mobile applications and that the automation of GUI testing remains a challenging task.

The main techniques proposed in literature to automate test cases generation and execution in the context of mobile applications can be classified in Capture/Replay, Model based, Model Learning and Random techniques [8].

Capture/Replay techniques record user interactions generated by human testers and convert them into test scripts that can be automatically replayed. Recent contributions in this field are the ones of Liu et al. [15] and White et al. [24]. Capture/Replay techniques need a remarkable effort to collect a sufficient number and variety of interactions from users or testers in order to obtain effective test suites. Moreover, test cases produced by different testers may be very similar between them, so the test suites may be quite redundant and inefficient. Model Based techniques are able to generate test cases from structural and/or behavioural models of the application. Examples of model based techniques in the field of mobile applications are the ones proposed in [23], [12], [26] and [4]. Model Learning techniques are able to build their own models by systematically exploring the behaviour of the GUI of a mobile application. They automatically generate test cases corresponding to sequences of triggered user and system events. In the context of mobile applications, recent Model Learning techniques have been proposed in [6], [17], [5]. Random testing techniques differ from Model Learning techniques because the sequence of triggered events is randomly chosen. Random techniques are very popular in the Android environment. In particular, Monkey [1] is a command line executable testing tool included

[1]http://developer.android.com/tools/help/monkey.html

DeMobile'15, August 31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3815-8/15/08...$15.00
http://dx.doi.org/10.1145/2804345.2804348

in the standard Android Development Toolkit (ADT) that is able to generate sequences of events on the interface of an Android application in a totally automatic manner. Other random techniques have been recently proposed in [10], [16] and [3]. The popularity of random techniques is due to their suitability to every application without the need for any specific knowledge about them and to the good level of effectiveness that they are able to reach. Unfortunately, random techniques generate test cases that are redundant, inefficient and very difficult to comprehend and manage (e.g. for debugging purposes).

Search based testing techniques represent a promising trade-off between Model Learning and Random techniques to generate effective and efficient test suites. Search Based Software Testing [1] is a specialization of Search Based Software Engineering (SBSE) [11] [9] related to the application of metaheuristic techniques to the problem of automatic generation of test cases optimizing the fault finding or the code coverage with a reasonable effort. In particular, in the set of metaheuristics algorithms, genetic algorithms are often used [2]. Genetic algorithms try to imitate the natural process of evolution: a population of candidate solutions, called chromosomes (i.e. test cases) is evolved using search operators such as selection, crossover, and mutation, gradually improving the fitness value of the individuals, until an optimal solution has been found or the search is stopped after a fixed time or a fixed number of evolutions.

At the best of our knowledge, only a single contribution related to the application of Search Based techniques to mobile application GUI testing can be found in literature. Mahmood et al. [17] present a search based technique supported by the EvoDroid tool for evolutionary testing of Android applications. EvoDroid automatically extracts two static models of the application under test, i.e. the Interface Model and the call Graph Model and generates evolutionary tests on the basis of these models.

In this paper we propose a novel search based testing technique called AGRippin (that is an acronym for Android Genetic Ripping) applicable to Android applications with the purpose to generate test suites that are both effective in terms of coverage of the source code and efficient in terms of number of generated test cases. Our technique is based on the combination of genetic and hill climbing algorithms and presents some peculiar contributions. In details, we propose:

- a representation based on a GUI test model that is able to describe test cases in form of *chromosomes* and actions on GUI interfaces in form of *genes*;
- a single cut crossover technique that is able to generate re-executable test cases;
- a mutation technique based on input equivalence classes;
- a fitness metric based on the measure of the diversity between the code coverage provided by the test cases;
- a technique to combine the genetic algorithm with a hill climbing algorithm in order to increase the rapidity of the algorithm.

The proposed technique has been implemented by a tool extending our previously presented Android Ripper tool [5] and tested in a case study involving five open source applications published on Google Play. Our approach differs from the EvoDroid one because it is completely automated and it does not rely on any previous knowledge or model of the application under test.

The paper is organized in the following way: section 2 reports a description of the the proposed technique while section 3 shows the results obtained of the case study. Finally, conclusive remarks and future works are proposed in section 4.

2. THE AGRIPPIN TECHNIQUE

In this section AGRippin, a Search Based technique for the generation of test cases optimizing the effectiveness in terms of coverage of the source code of the applications under test, is described.

According to the terminology of genetic algorithms, the *solution* proposed by the algorithm is, at each iteration, an evolved test suite that is composed of a *population* of *chromosomes* corresponding to test cases. Each chromosome is composed of *genes* corresponding to basic interactions with the application under test (AUT).

The effectiveness η of a test suite T is measured (in percentage) as the fraction of lines of source code (LOCs in the following) of the AUT covered by at least one of the test cases composing the test suite generated by the algorithm. It can be evaluated by the following formula:

$$\eta(T) = 100 * \frac{|\bigcup_{t \in T} Cov(t)|}{|LOC|}$$

where $t \in T$ is a test case included in the test suite T, $Cov(t)$ is the set of lines of code that is covered by the test case t and LOC is the set of lines of code of the AUT.

The efficiency ϵ of a test suite T can be defined as the ratio between its effectiveness and the number of generated test cases:

$$\epsilon(T) = \frac{\eta(T)}{|T|}$$

Our technique adopts a constraint of genetic algorithms for which the size of the population is constant at each iteration and is equal to the size of the initial population. Due to this constraint, the maximization of the effectiveness implies the maximization of the efficiency.

In the next subsection are described the characteristics of our technique in terms of chromosome representation, metrics for fitness evaluation, techniques for crossover, mutation, selection, and combination with a hill climbing technique.

2.1 Representation

The test suites generated by our technique are composed of test cases that are sequences of interactions with the GUI of the AUT. The application GUIs are abstracted according to the conceptual model shown in Figure 1.

According to this model, the GUI is composed of instances called *GUI Interfaces*; a GUI Interface is composed of a set of visual items called *Widgets*; each Widget is defined by a *Type* and some *Properties* with their *Names* and *Values*. Examples of Widget properties are its position on the screen, its identifier and so on. *Event Handlers* are methods that can be defined in the context of a GUI Interface or directly in the context of a Widget and that are executed in response to the occurrence of an *Event*. Events may be *User Events* if they are triggered by a user interaction on the GUI (e.g.

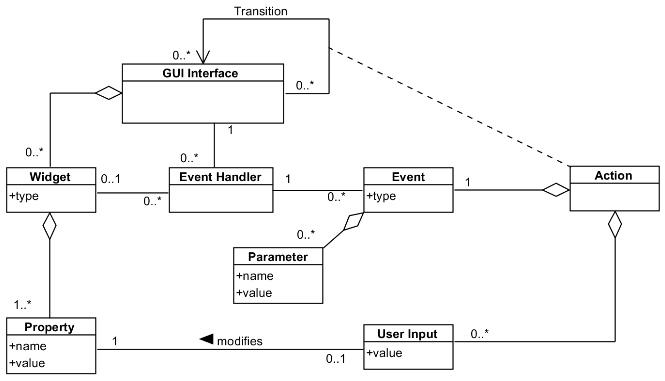


Figure 1: Conceptual Model of a GUI Interface

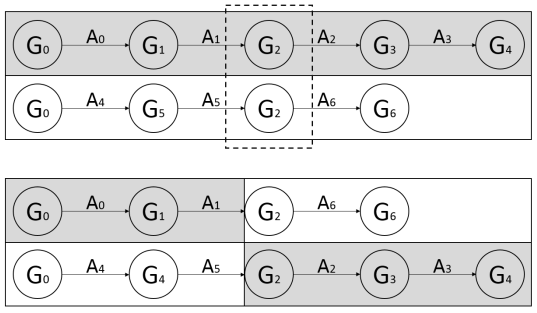


Figure 2: Crossover Example

the tap on a button), or *System Events* if they are triggered by the execution environment (e.g. a pausing of the application). An Event may have zero or more *Parameters*; each Parameter is identified by a *Name* and a *Value*. As an example, parameters of a tap event are the coordinates of the point of the GUI Interface where the tap is performed by the user. An *Action* is composed by an Event and one or more *User Inputs* and is able to trigger a *Transition* between two GUI Interface instances (not necessarily different between them). An User Input consists of the modification of a Value of a property (e.g. the insertion of a text in a editable text field) that does not cause the execution of any event handler (elsewhere it is modeled as an Action).

On the basis of these definitions, a *Test Case t* is a sequence of pairs $\{G, A\}$, where A is an Action that can be performed on the GUI Interface G and that may generate the GUI Interface of the next pair of the sequence. The first pair starts from the Home interface of the app G_0. A Test Case can be also defined as $t = (G_0, A_0, ...\ G_m, A_m)$, where G_0 is the Home interface. A set of such sequences is a *Test-Suite*.

2.2 Crossover

The crossover operator we exploited in our implementation is a Single-Point Crossover. Given two Test-Cases t_1 and t_2, the operator randomly chooses two pairs $\{G_i, A_i\} \in t_1$ and $\{G_j, A_j\} \in t_2$ and operates the crossover operation as shown in Figure 2.

A problem of this crossover operator is that it may generate sequences that do not correspond to executable test cases. If the generated test cases cannot be executed, they have to be discarded and the crossover operator has to be repeated until it generates a pair of executable test cases. In order to reduce the occurrence of such non-executable test cases, we propose a technique to candidate pairs of test cases and cut points for which the crossover operator should be applicable, based of two heuristic criteria of equivalence between GUI interfaces and between actions. The two heuristic criteria of equivalence are defined in the following ways:

EC1 Two GUI interfaces are considered equivalent if they include the same set of widgets and they define the same set of event handlers.

EC2 Two actions are considered equivalent if they are associated to the same user actions and the same event.

Let's consider two test cases $t_1 = (G_0, ..., G_i, A_i, ...)$ and $t_2 = (G_0, ..., G_j, A_j, ...)$ having the same starting GUI interface G_0. The pairs (G_i, A_i) and (G_j, A_j) are a candidate crossover point for our heuristic technique if they satisfy all these four criteria:

C1 the two GUI interfaces G_i and G_j are equivalent according to the $EC1$ criterion;

C2 the two actions A_i and A_j are not equivalent according to the $EC2$ criterion;

C3 the subsequence of t_1 which precedes the GUI interface G_i and the subsequence of t_2 which precedes the GUI interface G_j are not composed of a sequence of GUI interfaces and actions that are all respectively equivalent (according to the two criteria $EC1$ and $EC2$), *and* they are not both empty;

C4 the subsequence of t_1 which follows the action A_i and the subsequence of t_2 which follows the action A_j are not composed of a sequence of GUI interfaces and actions that are all respectively equivalent (according to the two criteria $EC1$ and $EC2$), *and* they are not both empty.

It's interesting to note that the first criterion avoids to select crossover points for which the action A_j is not applicable to the GUI interface G_i or the action A_i is not applicable to the GUI interface G_j. The other three criteria avoids the selection of crossover points that generate two test cases that are too similar or identical to the original ones.

As an example, let's observe the crossover example in Figure 2, in which equivalent GUI interfaces are labeled with the same label. We can verify that the selected crossover point (corresponding to the pairs (G_2, A_2) and (G_2, A_6)) is the unique one that satisfies all the four criteria (the pairs (G_0, A_0) and (G_0, A_4) satisfy the first two and the fourth criterion but they do not satisfy the third criterion because they are both preceded by an empty sequence).

The crossover points are randomly chosen in the set of the ones that satisfy these criteria. The test cases t_1 and t_2 are not removed from the test suite after the execution of the crossover operator, in coherence with the techniques proposed in the steady state genetic algorithms [22] (in example the Genitor one proposed by Whitley [25]). These techniques cause an increase in the population size that is restored to its initial size by the selection operator that is presented in the following. The crossover operator may be executed multiple times in the same iteration of the algorithm. We define the *crossover ratio* as the ratio between the number of test cases generated by the crossover at each iteration and the number of test cases of the initial solution.

2.3 Mutation

The mutation operator we proposed in this technique modifies the Actions by mutating the values of the User Inputs or the Event Parameters values. In each mutation, the value of a single parameter of the action is changed to a new value belonging to a static set of equivalence classes according to the parameter type. In example, the value of an editable text field may be set to a random string, a number or a correct email address while the location parameter of a GPS event may be set to coordinates values over or under the equator. The new test obtained after a mutation could not be executable if the GUI interface reached after the execution of the mutated action is not equivalent to the one reached by the original action. In this case, we consider that the new mutated test case terminates with the mutated action and the new test case is shorter than the original one.

The mutation operator randomly selects the test case and the action to be mutated in all the test suite. Mutated test cases are added to the test suite and the original ones are not removed. We define the *mutation ratio* as the ratio between the number of test cases generated by the mutation operator and the number of test cases of the initial solution.

2.4 Fitness Evaluation

We defined two distinct Fitness measures: the Global Fitness (that is the effectiveness η of the generated test suite and is measured in terms of the code coverage reached by the test cases of the test suite as described above in this section), and the Local Fitness expressing the degree of diversity of a single test case with respect to the set of the test cases of the test suite.

The Local Fitness measure ranks the individuals in terms of their potential contribution to the Global Fitness of the solution and of their diversity. To this aim, we propose a *rank* measure that is able to order all the test cases of the test suite. The Local Fitness measure is composed of two components named $F1$ and $F2$. The first component $F1$ the following three values, in order of decreasing rank:

- L_1, if the test case covers one or more lines that are not covered by any other test case;
- L_2, if the test case has a coverage set that (i) includes only lines of code that are covered by at least another test case of the test suite but that (ii) is not included in the set of lines covered by any other test case of the current test suite;
- L_3, if the test case has a coverage set that is included in the coverage set of at least another test case of the solution.

As regards the set of test cases having the same coverage set, the algorithm conventionally assigns a L_2 value to one test case (randomly selected) of the set and the L_3 value to all the other test cases of the set. Intuitively, test cases having a L_1 value are the ones that should be preserved to avoid a sure loss of effectiveness, whereas test cases having a L_3 value are the better candidates to be filtered out by the selection operator.

The second component $F2$ of the Local Fitness represents a weighted measure of code coverage and is defined by the following formula:

$$F2(t) = \sum_{l \in Cov(t)} w(l)$$

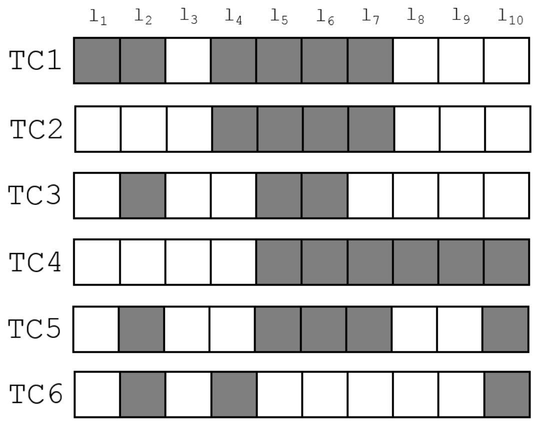


Figure 3: Test-Case Fitness Evaluation

where:

- $Cov(t)$ is the set of lines of code that are covered by the test case t
- $w(l)$ represents the relative weight of the coverage of the line l. It is defined as:

$$w(l) = \frac{1}{\sum_{u \in T} c(u)}$$

where $c(u) \in \{0, 1\}$. It is 0 if $l \notin Cov(u)$, 1 elsewhere.

$F2$ gives a measure of the relative importance of the coverage provided by a test cases in the context of a test suite because the coverage of lines that are covered by few test cases has a higher weight than the coverage of lines covered by many test cases. $F2$ is used to order between them test cases having the same $F1$ values.

As an example, let's consider the test suite shown in Figure 3 in which there is an AUT composed of 10 LOCs labeled as $\{l_1, ..., l_{10}\}$ and a Test Suite T = $\{TC1, ..., TC6\}$. The coverage of each test case is depicted in Figure 3 where black boxes corresponds to covered lines whereas white boxes corresponds to uncovered lines. In order to evaluate the Local Fitness we assigned $F1(TC1) = L_1$ and $F1(TC4) = L_1$ because they are respectively the unique test cases covering the line l_1 and the two lines l_8 and l_9. The values of $F1(TC2)$ and $F1(TC3)$ are instead set to L_3 because their coverage sets are respectively included in the ones of $TC1$ and $TC5$. The $F1$ value of the remaining test cases (i.e. $TC5$ and $TC6$) is set to L_2. In order to evaluate the $F2$ values for each test case, the weights w of each line l have to be evaluated. In example, the weight of line l_1 is 1 because it is covered exactly by one test case, whereas $w(l_2) = \frac{1}{4}$ because the line l_2 is covered by four test cases and so on. The Fitness Function of the test case $TC1$ is then equal to:

$$F2(TC1) = w(l_1) + w(l_2) + w(l_4) + w(l_5) + w(l_6) + w(l_7) = \frac{1}{1} + \frac{1}{4} + \frac{1}{3} + \frac{1}{5} + \frac{1}{5} + \frac{1}{4} = 2.23$$

The $F2$ values for each test case are reported in Table 1. In this table the test cases are ordered for decreasing values of $F1$ and, for test cases with the same $F1$ value, for decreasing values of $F2$. The $RANK$ column expresses the ordering position between all the test cases of the test suite.

Table 1: Test-Case Classification Example

RANK	**t**	$F1$	$F2$
1	TC4	L_1	2.98
2	TC1	L_1	2.23
3	TC5	L_2	1.23
4	TC6	L_2	0.91
5	TC2	L_3	0.98
6	TC3	L_3	0.65

2.5 Selection

The selection operator restores the size of the test suite to its initial value (corresponding to the size of the initial test suite) by deleting the test cases having the worst values of Local Fitness in coherence with the rank selection operator firstly proposed by Baker [7].

The fraction of test cases that are selected for deletion at each iteration is named *turnover ratio* and it is the sum of the *crossover ratio* and of the *mutation ratio*. As an example, if the crossover ratio is 1/3, then the test cases TC2 and TC3 shown in Table 1 have to be deleted.

2.6 Combination Technique

By means of the application of the crossover and of the mutation operator GUI interfaces that are not equivalent to any of the already visited ones may be discovered. These GUI interfaces have not yet been visited by any test case and they contains different sets of widgets and event handlers with respect to the other ones. This represents a positive achievement in terms of global fitness because code that has not yet been covered and that corresponds to the execution of these event handlers may now be executed.

In the AGRippin technique we propose a *combination* of the genetic technique with a model learning technique that will be started only when a new GUI interface is discovered. This technique aims at the systematic generation of new test cases including at least an event of each new discovered GUI interface and is very similar to the one we have proposed in the past [5]. This technique can be seen as a Hill Climbing technique because it selects at each iteration the most promising sequences, i.e. the ones in which at least a new line, corresponding to a new event handler call is covered. In order to restore the size of the test suite to its initial value, a re-execution of the selection operator has to be carried out after each execution of the model learning technique.

The adoption of hybrid algorithms combining genetic and hill climbing algorithms have been already presented in literature, with good results [18] and some criticism. We adopted this solution for two reasons: (1) because our specific implementation of the mutation operator is not able to generate new events but only to mutate their parameters and (2) to accelerate the process of exploring the interactions related to portions of the application that are discovered but not explored by crossover and mutation operators.

3. CASE STUDY

This section reports the results of some case studies that we carried out with the aim to assess the effectiveness of the proposed search based testing technique. We have implemented the technique in the context of Android applications and we have applied it to five open-source Android applications.

We have compared the test suites generated by our technique with the ones generated by the Android Ripper Tool [5] that we developed in the past. It realizes a Model Learning technique for the exploration of the GUI of Android applications. Since the Android Ripper explores at each iteration the GUIs of an Android application by executing an event that have never been executed before, we can consider this technique as a kind of Hill Climbing technique because an increment in code coverage is surely expected by the execution of each new event.

The purpose of our experimentation is to provide an answer for the following research question:

RQ: Are the test suites generated by the proposed technique more effective than the ones generated by the considered Hill Climbing technique?

The effectiveness of the generated test suites is measured (in percentage) as the fraction of lines of code of the AUT that are covered at least once by at least a test case of the test suite T:

$$\eta(T) = 100 * \frac{|\bigcup_{t \in T} Cov(t)|}{|LOC|}$$

3.1 Subjects

Five real-world open source Android Applications have been selected for our study; they are all published and freely available on the Google Play market. Some details about these application are reported in Table 2. They are all medium sized applications, with a number of LOCs varying from 2308 lines (AUT1) to 6770 lines (AUT3).

Application preconditions can affect the effectiveness of the generated test cases [5]. For the purpose of this experimentation, we chosen the same set of preconditions for each application and used it in all the experiments with both the techniques (as an example, for AUT1 we preloaded the same dictionary in the SD card before the execution of each test case).

3.2 Experiment Environment and Setup

The experimentation has been carried out by using two tools that we have implemented, i.e. the Android Ripper tool and the AGRippin tool.

The Android Ripper tool [2] [5] has been used to systematically explore the GUIs of Android applications with a breadth-first strategy. Each branch of the exploration carried out by the Android Ripper tool is terminated when a GUI interface is found that is equivalent (in the sense that it has the same widgets and event handlers) to a previously visited one. The Android Ripper tool produces a test suite composed of test cases corresponding to the explored execution paths.

The Android Ripper tool is composed of two main components. The *Driver* component is responsible for the execution of the exploration algorithm and for the generation of the resulting test cases in form of Android JUnit test cases exploiting the Robotium library [3]. The *Device* component is deployed and executed in the context of an Android emulator and is able to execute actions on the AUT, to extract the obtained GUI interfaces and to send their description to the *Driver* component via the Android Debug Bridge (ADB)[4]

[2]https://github.com/reverse-unina/AndroidRipper
[3]https://code.google.com/p/robotium/
[4]http://developer.android.com/tools/help/adb.html

Table 2: Android Applications (AUTs)

	Application	Description	Link	LOCs	Activities
AUT1	AardDict 1.4.1	A dictionary application	https://github.com/aarddict/android	2308	7
AUT2	TomDroid 0.7.1	A manager for notes	https://code.launchpad.net/tomdroid	4167	10
AUT3	OmniDroid 0.2.1	A manager for device automated tasks and actions	https://code.google.com/p/omnidroid/	6770	16
AUT4	AlarmClock 1.7	An alarm clock	https://code.google.com/p/kraigsandroid/	2320	5
AUT5	BookWorm 1.0.18	A manager for book collections	https://code.google.com/p/and-bookworm/	3190	10

utility. The code coverage has been measured by means of the Emma utility [5] included in the Android SDK.

The AGRippin tool[6] has been realized on top of the Android Ripper tool by implementing an *AGR* component responsible of the execution of the proposed technique. The *AGR* component interacts with both the components of the Android Ripper tool.

The experimentation has been carried out on 6 different Intel I5 PCs with a clock frequency of 3.0GHz, 4GB of RAM and Windows 7 64bit operative system. On these machines we installed an Android Virtual Device [7] (AVD) emulating the Android Gingerbread 2.3.3 operative system, with 512MB of RAM and an emulated 64MB SD Card.

We started the experimentation by carrying out an exploration of each AUT by means of the Android Ripper tool that generated a test suite. We considered these test suites as a result of an Hill Climbing exploration because the Ripper strategy consists of the selection, at each step, of the most promising action, i.e. of an action that has not been previously executed. The test suite generated by the Android Ripper tool has been used, too, as the initial solution of the search based testing technique.

The AGRippin technique has been configured in our experimentation by fixing the parameters shown in Table 3. The *Crossover ratio* and the *Mutation ratio* respectively represent the fraction of the test cases of a test suite that are involved in a crossover or in a mutation at a given iteration. In accordance with the suggestions of Mitchell et al. [19] we set a higher value for the Crossover ratio with respect to the Mutation ratio. The *Number of Iterations* represents the termination condition of our algorithm in terms of the number of performed iterations. We fixed an arbitrary value of 30 in this experimentation. In order to take into account the randomness of our search based testing technique, we executed the AGRippin technique six times with six different seeds for any AUT.

Table 3: Configuration Parameters Values

Parameter	Value
Crossover ratio	20%
Mutation ratio	5%
Number of Iterations	30

3.3 Results and Discussions

Table 4 reports the results obtained by the execution of our experimentation on the five AUTs.

The first column of the table reports the effectiveness $\eta(T_0)$ of the test suite T_0 generated by the Hill Climbing (HC) technique implemented by the Android Ripper tool. The columns labeled $\mu(\eta(T))$ and $\sigma(\eta(T))$ respectively report the average and the standard deviation of the effectiveness η of the test suites T generated by the AGRippin technique (abbreviated in AGR in Table 4) in six different executions featuring different random seeds. The fourth column of the table reports the maximum number of interfaces discovered by AGRippin that have not been discovered by HC. The fifth column reports the number of test cases composing both the test suites generated by HC and AGRippin. Finally, the last two columns respectively report the HC execution time and the average execution time of AGRippin (after 30 iterations), measured on the same machines. The results in this table show that for all the considered AUTs the AGRippin technique is able to provide an increase in coverage with respect to the Hill Climbing technique that varies from 1% (for AUT4) to 24% (for AUT1), so we can conclude that the proposed RQ has a positive answer. As regards the execution time, we can observe that the average time needed to execute 30 iterations with AGRippin varies from 6 to 12 times the amount of time needed to execute HC. The values of standard deviation $\sigma(T)$ show that the effectiveness of AGRippin depends on the randomness in a remarkable way. We can hypothesize that the execution of a larger number of parallel sessions can provide improvements in the effectiveness of the AGRippin technique without increasing the execution time.

In order to show an example of the dependence of the effectiveness on the number of iterations, Figure 4 reports the effectiveness trends observed for the Bookworm application (AUT5). The figure shows the trends of the coverage of the test suites obtained by six different executions (with six different random seeds) of the AGRippin technique (named AGR1, AGR2, AGR3, AGR4, AGR5, AGR6 in figure) as the number of iterations increases. The dashed line shown in figure represents the coverage percentage provided by the test suite generated by the HC technique. We can observe that each execution of AGRippin constantly reaches higher values of effectiveness than HC. In the Bookworm application, one interface has been discovered by each AGRippin execution and in these cases a large portion of unexecuted code have been systematically been explored by the Hill Climbing technique. We can note its effect by observing the rapid rising in the effectiveness that occurs once for each AGRippin execution. The values of effectiveness after 30 iterations are slightly different between them. On the basis of this phenomenon we can hypothesize that the execution of a larger number of iterations may provide better results and a reduced dependency on randomness. Similar considerations can be done for all the other AUTs.

In order to provide more details about the capability of the AGRippin technique to cover unexplored portions of code, we examined in details the differences in coverage between the executions of the HC and AGRippin techniques. We recognized improvements in coverage due to the Crossover

[5]http://emma.sourceforge.net/

[6]https://github.com/reverse-unina/agrippin

[7]https://developer.android.com/tools/devices/index.html

Table 4: Experimental Results

	HC	AGR				Execution Time (hours)	
	$\eta(T_0)$	$\mu(\eta(T))$	$\sigma(\eta(T))$	Discovered Interfaces	$\|T\|$	HC	AGR
AUT1	43.07%	67.10%	0.26%	1	51	3.5	22
AUT2	28.08%	32.61%	2.45%	0	51	2.5	30
AUT3	51.58%	58.31%	2.28%	0	162	6	55
AUT4	66.90%	68.00%	1.21%	0	68	2.8	20
AUT5	40.34%	47.22%	0.45%	1	50	3.2	23

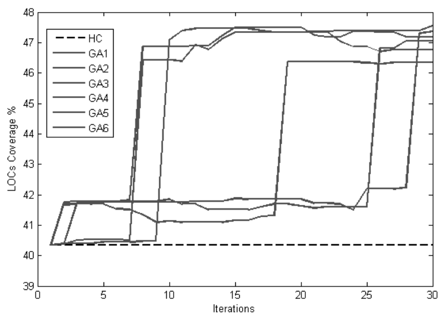


Figure 4: Effectiveness Trends for AUT5

operator, to the Mutation operator and to the Combination technique.

As regards the improvements due to the crossover operator, we observed that in some cases the mixing of two different test cases produced new execution sequences that are able to show different behaviors of the AUT. As an example, for AUT2 the crossover operator generates a new test case executing the backup functionality after the changing of its settings causing the execution of a different backup scenario. Another example of unexplored code executed by a crossover operator is, in AUT5, the one related to the sequential execution of the insertion of a book in the book list followed by the visualization of this book list. In the test suite generated by HC, the visualization was executed only with an empty book list whereas AGRippin were able to visualize book lists that are not empty.

As regards the improvements in effectiveness due to the mutation operator, we report two exemplar cases. An AUT1 functionality is the search in a vocabulary of a string included in an input text field. Whereas the HC technique only provided a random text to this input field, the mutation operator implemented in AGRippin generated a new test case with an input text value belonging to the equivalence class of the English words. This mutation caused the execution of a portion of code related to the retrieval of the word in the dictionary and to the visualization of one undiscovered interface showing the list including one or more search results. Another example is the one found in AUT4, where the insertion of an input value belonging to the negative numbers equivalence class in a specific text field caused the execution of a portion of code executing a validating check that has not been tested by HC.

Finally, the combination technique has been applied in two cases to explore two interfaces discovered in AUT1 and AUT5 (corresponding to two of the cases described above). In these cases the application of the HC technique on these interfaces caused a great improvement in code coverage (about 5% of improvement in both the cases).

4. CONCLUSIONS AND FUTURE WORKS

In this paper a novel search based testing technique has been proposed and implemented in the context of Android applications. We evaluated its effectiveness by carrying out a case study involving five open source Android applications. We demonstrated that the technique is more effective than an Hill Climbing technique based on the systematic exploration of the GUI events executable on an Android application, in terms of source code coverage.

In order to generalize the promising results shown by this preliminary experimentation, in future we will extend our experimentation to a larger set of Android applications and we will extend our comparative analysis to random techniques and other techniques for automatic test case generation proposed in literature.

One of the objectives of the future experimentation will be the tuning of the algorithm parameters values (such as the crossover ratio, the mutation ratio, the number of iterations and the test suite size) and the evaluation of their influence on the effectiveness of the generated test suites and on the number of iterations needed to reach this level of effectiveness. As regards the crossover and mutation ratio, we plan to implement a technique based on adaptive variations (as the one proposed by Srinivas and Ptnaik in [21]) in order to reduce the probability that the generated test suites maintain the same coverage for many consecutive iterations, as experienced in some of the case studies. As regards the number of iterations we will carry out longer experiments in order to evaluate if some phenomena of convergence of the coverage to a global maximum may be observed. Finally, we plan to test a variant of the technique in order to increase the test suite size when new test cases are generated by the Hill Climbing technique in order to avoid the loss of important test cases due to the selection operator.

5. ACKNOWLEDGMENTS

The authors would like to thank Carmine Cirillo for his contribution to the implementation of some components of the Agrippin tool.

6. REFERENCES

[1] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Inf. Softw. Technol.*, 51(6):957–976, June 2009.

[2] S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on*, 36(6):742–762, Nov 2010.

[3] D. Amalfitano, N. Amatucci, A. R. Fasolino, P. Tramontana, E. Kowalczyk, and A. Memon. Exploiting the saturation effect in automatic random testing of android applications. In *The Proceedings of the 2nd ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft 2015)*, 2015.

[4] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, and A. Memon. Mobiguitar – a tool for automated model-based testing of mobile apps. *Software, IEEE*, PP(99):1–1, 2014.

[5] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *Proc. of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

[6] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. *SIGPLAN Not.*, 48(10):641–660, Oct. 2013.

[7] J. Baker. Adaptive selection methods for genetic algorithms. volume Proceedings of the First International Conference on Genetic Algorithms and Their Applications. Erlbaum, 1985.

[8] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon. Graphical user interface (gui) testing: Systematic mapping and repository. *Information and Software Technology*, 2013.

[9] J. Clarke, J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *Software, IEE Proceedings -*, 150(3):161–175, June 2003.

[10] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. 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, New York, NY, USA, 2014. ACM.

[11] M. Harman and B. F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833 – 839, 2001.

[12] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 67–77, New York, NY, USA, 2013. ACM.

[13] M. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, pages 15–24, Oct 2013.

[14] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the test automation culture of app developers. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10, April 2015.

[15] C.-H. Liu, C.-Y. Lu, S.-J. Cheng, K.-Y. Chang, Y.-C. Hsiao, and W.-M. Chu. Capture-replay testing for android applications. In *Computer, Consumer and Control (IS3C), 2014 International Symposium on*, pages 1129–1132, June 2014.

[16] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[17] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 599–609, New York, NY, USA, 2014. ACM.

[18] H. Mühlenbein. How genetic algorithms really work: Mutation and hillclimbing. In R. Männer and B. Manderick, editors, *PPSN*, pages 15–26. Elsevier, 1992.

[19] M. Mitchell, S. Forrest, and J. H. Holland. The royal road for genetic algorithms: Fitness landscapes and ga performance. In *Proceedings of the First European Conference on Artificial Life*, pages 245–254. MIT Press, 1991.

[20] H. Muccini, A. Di Francesco, and P. Esposito. Software testing of mobile applications: Challenges and future research directions. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 29–35, June 2012.

[21] M. Srinivas and L. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *Systems, Man and Cybernetics, IEEE Transactions on*, 24(4):656–667, Apr 1994.

[22] G. Syswerda. A study of reproduction in generational and steady-state genetic algorithms. In G. J. Rawlins, editor, *Foundations of genetic algorithms*, pages 94–101. Morgan Kaufmann, San Mateo, CA, 1991.

[23] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based gui testing of an android application. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 377–386, Washington, DC, USA, 2011. IEEE Computer Society.

[24] M. White, M. Linares-Vásquez, P. Johnson, C. Bernal-Cárdenas, and D. Poshyvanyk. User guided automation for testing mobile apps. In *23rd IEEE International Conference on Program Comprehension, (ICPC'15)*, volume 1, page to appear, May 2015.

[25] D. Whitley and K. Kauth. GENITOR: A different genetic algorithm. In *Proceedings of the 1988 Rocky Mountain Conference on Artificial Intelligence*, pages 118–130, 1988.

[26] W. Yang, M. R. Prasad, and T. 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, Berlin, Heidelberg, 2013. Springer-Verlag.

Detecting Android Malware using Sequences of System Calls

Gerardo Canfora
Dept. of Engineering
University of Sannio
Benevento, Italy
canfora@unisannio.it

Eric Medvet
Dept. of Engineering and Architecture
University of Trieste
Trieste, Italy
emedvet@units.it

Francesco Mercaldo
Dept. of Engineering
University of Sannio
Benevento, Italy
fmercaldo@unisannio.it

Corrado Aaron Visaggio
Dept. of Engineering
University of Sannio
Benevento, Italy
visaggio@unisannio.it

ABSTRACT
The increasing diffusion of smart devices, along with the dynamism of the mobile applications ecosystem, are boosting the production of malware for the Android platform. So far, many different methods have been developed for detecting Android malware, based on either static or dynamic analysis. The main limitations of existing methods include: low accuracy, proneness to evasion techniques, and weak validation, often limited to emulators or modified kernels.

We propose an Android malware detection method, based on sequences of system calls, that overcomes these limitations. The assumption is that malicious behaviors (e.g., sending high premium rate SMS, cyphering data for ransom, botnet capabilities, and so on) are implemented by specific system calls sequences: yet, no apriori knowledge is available about which sequences are associated with which malicious behaviors, in particular in the mobile applications ecosystem where new malware and non-malware applications continuously arise. Hence, we use Machine Learning to automatically learn these associations (a sort of "fingerprint" of the malware); then we exploit them to actually detect malware. Experimentation on 20 000 execution traces of 2000 applications (1000 of them being malware belonging to different malware families), performed on a real device, shows promising results: we obtain a detection accuracy of 97%. Moreover, we show that the proposed method can cope with the dynamism of the mobile apps ecosystem, since it can detect unknown malware.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification -Validation; D.4.6 [**Operating Systems**]: Security and Protection—Invasive software

General Terms
Security

Keywords
malware, Android, dynamic analysis, security, machine learning

DeMobile'15, August 31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3815-8/15/08...$15.00
http://dx.doi.org/10.1145/2804345.2804349

1. INTRODUCTION
A number of surveys by specialised companies and commercial press articles provide evidence that the mobile malware on the Android platform is growing in volume and impact [6, 8, 7]. There are two general approaches to implement malware detectors based on (a) static analysis or (b) dynamic analysis. Broadly speaking, the former does not require many resources, in terms of enabling infrastructure, and is faster to execute than the latter, but it is more prone to be evaded with techniques whose effectiveness has been largely demonstrated in the literature [24, 30, 35]. The latter is harder to bypass, as it captures the behavior, but it usually needs more resources and cannot be run directly on the devices (it is often performed on virtual or dedicated machines).

We propose a malware detection technique based on dynamic analysis which considers sequences of system calls that are likely to occur more in malware than in non-malware applications. The rationale behind our choice can be explained as follows. Often, the process of malware evolution mainly consists of modifications to existing malware. Malware writers use to improve mechanisms of infection, obfuscation techniques, or payloads already implemented in previous malware, or tend to combine them [36]. This indeed explains why malware is classified in terms of families, i.e., malicious apps which share behaviors, implementation strategies and characteristics [10, 37]. As a consequence, malicious apps belonging to the same family are very likely to exhibit strong similarities in terms of code and behavior.

Our assumption is that the behavior similarities among malicious apps could be used to detect new malware. We chose to characterize behavior in terms of sequences of system calls as this representation is, on the one hand, specific enough to capture the app behavior and, on the other hand, it is generic enough to be robust to camouflage techniques aimed at hiding the behavior. In other words, we assume that the frequencies of a set of system calls sequences may represent a sort of *fingerprint* of the malicious behavior. Thus, new malware should be recognized when that fingerprint is found.

The contributions of this paper can be summarized as follows:

1. We designed a method for (i) automatically selecting, among the very large number of possible system calls sequences, those which are the most useful for malware detection, and, (ii) given the fingerprint defined in terms of frequencies of the selected system calls sequences, classifying an execution trace as malware or non-malware.

2. We performed an extensive experimental evaluation using a real device on which we executed 2000 apps for a total of 20 000 runs: we found that our method delivers an accuracy up to 97%. We remark that, in most cases, previous works based on dynamic analysis validated their proposals using emulators and modified kernels, which produce outcomes which are less realistic than the outcomes deriving from real devices, i.e., the one used in this experimentation.

3. We assessed our method also in the more challenging scenario of zero-day attacks, where the detection is applied to *new* malware applications or *new* malware families, i.e., to applications whose behavior has never observed before (and hence has not be exploited to build the fingerprint).

The reminder of the paper is organized as follows: Section 2 thoroughly analyses related work, Section 3 introduces the approach, and Section 4 illustrates the validation of the approach. Finally, Section 5 draws the conclusions.

2. RELATED WORK

Malware detection techniques can be characterized in terms of the features they use to discriminate between malware and non-malware applications: those features can be obtained by means of static analysis or dynamic analysis. In general, static analysis captures suspicious patterns within the code (or artifacts related to code, such as meta information), whereas dynamic analysis captures suspicious patterns related to the behavior observed during the application running [16, 28]. Here we review recent works based mainly on dynamic analysis, and specifically on the analysis of system calls [14, 19, 26, 32, 18, 27, 21, 31, 20, 13].

Canfora et al. [14] propose a method for detecting mobile malware based on three metrics, which evaluate: the occurrences of a reduced subset of system calls, a weighted sum of a subset of permissions which the application requires, and a set of combinations of permissions. The experimentation of this paper considers a sample of 200 real world malicious apps and 200 real world trusted apps and produced a precision of 74%. CopperDroid [26] recognizes malware through a system calls analysis using a customized version of the Android emulator able to tracking system calls. Wang et al. [33] use an emulator to perform a similar task. The method was validated on a set of 1600 malicious apps, and was able to find the 60% of the malicious apps belonging to one sample (the Genoma Project), and the 73% of the malicious apps included in the second sample (the Contagio dataset).

Jeong et al. [19] hook system calls to create, read/write operations on files, and intents activity to detect malicious behavior. Their technique hooks system calls and a binder driver function in the Android kernel. The authors used a customized kernel on a real device, and the sample included 2 malicious apps developed by the authors. In [32] the authors characterize the response of an application in terms of a subset of system calls generated from bad activities in background when they stimulate apps with user interfaces events. They use an Android emulator for running experiments and their evaluation is based on 2 malicious samples of DroidDream family (Super History Eraser and Task Killer Pro). Schmidt et al. [27] used the view from Linux-kernel such as network traffic, system calls, and file system logs to detect anomalies in the Android system. The method presented in reference [18] consists of an application log and a recorder of a set of system calls related to management of file, I/O and processes. They use a physical device, with an Android 2.1 based modified ROM image. The evaluation phase considers 230 applications in greater part downloaded from Google Play and the method detects 37 applications which steal some kinds of personal sensitive data, 14 applications which execute exploit code and 13 destructive applications.

Concerning system call usage in systems other than Android, authors in [21] developed a system calls collection module that was installed on ten machines running Microsoft Windows XP, used by people carrying out their normal activities. They observed 242 trusted different applications, collecting 1.5 billion system calls over a period of several weeks. Kolbitsch et al. [20] track dependencies among system calls to match the activity of an unknown program against the trained behavior models from six different malware families. They evaluate the technique with 300 Microsoft Windows worm with a detection effectiveness of 64% (varying from 10% regarding the Agent family from 90% with the Allaple family). In [13] the authors adopt system calls to detect malicious JavaScript. They evaluate two techniques for detecting malicious web pages based on system calls: the first one consists of counting the occurrences of specific system calls, while the second one consists of retrieving system calls sequences. The first technique produces an accuracy slightly higher that the second one (97% vs. 96%).

In summary, the main differences between our work and other dynamic analysis techniques for malware detection are:

- Our approach takes into account all the system calls rather than a reduced set and we consider sequences of system calls, rather than system calls taken in isolation.

- We validate our approach on a large set of 2000 applications. The only papers which consider a very large dataset [17, 38] do not propose neither assess a detection method, but evaluate properties of the infection rate within a specific marketplace.
- We obtain a high detection accuracy (97%). The papers proposing a method with better performances than ours have a data set whose size is much smaller than our data set and, in some cases, malware samples are written by the authors rather than taken from the real marketplaces.
- We run the experimentation on a real device, while quite all the papers make use of emulators, which reduces the truthfulness of the experiments.

3. DETECTION METHOD

We consider the problem of classifying an execution trace of a mobile application as trusted or malicious, i.e., classifying the corresponding application as non-malware or malware. An *execution trace* is a sequence t in which each element represents a system call being issued by the application under analysis (AUA): we consider only the function name and discard all the parameters values. We denote by C the set of all the possible system calls, i.e., the alphabet to which symbols forming a sequence t belong.

A system call is the mechanism used by a user-level process or application layer to request a kernel level service to the operating system, such as power management, memory, process life-cycle, network connection, device security, access to hardware resources [29]. When performed, a system call implies a shift from user mode to kernel mode: this allows the kernel of the operating system to perform sensitive operations. When the task carried out by the invoked system call is completed, the control is returned to the user mode. In [9] Android kernel invocations are sub-grouped into: (i) system calls which directly invoke the native kernel functionality; (ii) binder calls, which support the invocation of Binder driver in the kernel (Binder is a system for inter-process communication); and (iii) socket calls, which allow read/write commands and send data to/from a Linux socket. System calls are not called directly by a user process: they are invoked through interrupts, or by means of asynchronous signals indicating requests for a particular service sent by the running process to the operating system. A Linux kernel (which the Android system builds on) has more than 250 system calls: our method considers all the system calls generated by the AUA when running.

The classification method here proposed consists of two phases, the training phase, in which a classifier is built on a set of labeled examples, and the actual classification phase, in which a trace is classified by the classifier.

In the training phase, we proceed as follows. Let T be a set of *labeled examples*, i.e., a set of pairs (t, l) where t is a trace and $l \in \{\text{trusted}, \text{malicious}\}$ is a label. We first transform each trace t in a feature vector $\mathbf{f}(t) \in [0,1]^{|C|^n}$, where n is a parameter. Each element of $\mathbf{f}(t)$ represents the relative occurrences of a given n-long subsequence of system calls in t. For example, let $n = 3$ and let $f_{2711}(t)$ represents the relative occurrences of the subsequence $\{\texttt{open}, \texttt{stat64}, \texttt{open}\}$ in t (i.e., in this example, the index of the element of $\mathbf{f}$ which corresponds to the subsequence $\{\texttt{open}, \texttt{stat64}, \texttt{open}\}$ is 2711), then $f_{2711}(t)$ is obtained as the number of occurrences of that subsequence divided by the sequence length $|t|$.

The number $|C|^n$ of different features may be remarkably large: for this reason, we perform a feature selection procedure aimed at selecting only the k feature which best discriminate between trusted and malicious applications in T. The feature selection procedure comprises of two steps. Let T_t and T_m be respectively the set of trusted and malicious traces, i.e., $T_t := \{(t,l) \in T : l = \text{trusted}\}$ and $T_m := \{(t,l) \in T : l = \text{malicious}\}$.

In the first step, we compute, for each ith feature, the relative class difference δ_i as:

$$\delta_i = \frac{\left| \frac{1}{|T_t|} \sum_{(t,l)\in T_t} f_i(t) - \frac{1}{|T_m|} \sum_{(t,l)\in T_m} f_i(t) \right|}{\max_{(t,l)\in T} f_i(t)}$$

We then select the $k' \gg k$ features with the greatest δ_i among those for which $\max_{(t,l)\in T} f_i(t) > 0$—$k$ and k' are parameters of the method.

In the second step, we compute, for each ith feature among the k' selected at the previous step, the mutual information I_i with the label. We then select the k features with the greatest I_i. During the exploratory analysis, we also experimented with a feature selection procedure which took into account, during the second step, the inter-dependencies among features: we found that it did not deliver better results.

Finally, we train a Support Vector Machine (SVM) on the selected features for the traces contained in T: we use a Gaussian radial kernel with the cost parameter set to 1.

The actual classification of an execution trace t is performed by computing the corresponding feature vector $\mathbf{f}(t)$, retaining only the features obtained by the aforementioned feature selection procedure, and then applying the trained classifier.

4. EXPERIMENTAL EVALUATION

We performed an extensive experimental evaluation for the purpose of assessing our method in the following detection scenarios:

- *Unseen execution trace* Ability to correctly classify an execution trace of an application for which other execution traces where available during the training phase.
- *Unseen application* Ability to correctly classify an execution trace of an application for which no execution traces were available during the training phase, but belonging to a malware family for which some traces were available during the training phase.
- *Unseen family* Ability to correctly classify an execution trace of a malware application belonging to a family for which no execution traces were available during the training phase.

Clearly, the considered scenarios are differently challenging (the former is the least challenging, the latter is the most challenging), since the amount of information available to the training phase, w.r.t. the AUA, is different. We investigated these scenarios using a single dataset $\mathcal{T}$, whose collection procedure is described in the next section, and varying the experimental procedure, i.e., varying the way we built the training set T and the testing set T' from $\mathcal{T}$.

4.1 Baseline

In order to provide a baseline, we designed and evaluated a detection method based on application permissions. As discussed in sections 2, permissions have been shown to be a relevant feature for the purpose of discriminating between malware and non-malware applications [10, 23, 22, 11, 14, 34, 15, 12].

In particular, in the baseline method we build a feature vector $\mathbf{f}$ for each application where an element f_i is 1 or 0, depending on the respective ith permission being declared for that application. The list of all the permissions (and hence the length of the feature vectors) is determined once in the training phase. The remaining part of the baseline method is the same as in the proposed method: two steps feature selection, SVM training and actual classification using the trained SVM on the selected features. Note that the only parameter which matters in the baseline method is the number k of selected features.

4.2 Data collection

4.2.1 Applications

We built a dataset of traces collected from 2000 Android applications, 1000 trusted and 1000 malware.

The trusted applications were automatically collected from Google Play [1], by using a script which queries an unofficial python API [3] to search and download apps from Android official market. The downloaded applications belong to different categories (call & contacts, education, entertainment, GPS & travel, internet, lifestyle, news & weather, productivity, utilities, business, communication, email & SMS, fun & games, health & fitness, live wallpapers, personalization). The applications retrieved were among the most downloaded in their category and they are free. We chose the most popular apps in order to increase the probability that these apps were actually trusted. The trusted applications were collected between January 2014 and April 2014 and they were later analysed with the VirusTotal service [5]. This service run 52 different antivirus software (e.g., Symantec, Avast, Kasperky, McAfee, Panda, and others) on the app: the output confirmed that the trusted apps included in our dataset did not contain malicious payload.

The malware dataset was obtained from the Drebin dataset. This dataset consists of a total of 5560 applications belonging to 179 different families, classified as malware [10, 30]. To the best of our knowledge, this is the most recent dataset of mobile malware applications currently used to evaluate malware detectors in literature.

The malware dataset includes 28 different families, unevenly represented in the dataset. In order to improve the validity of the experiment, we randomly selected 1000 applications.

4.2.2 Execution traces

We aimed at collecting execution traces which were realistic. To this end, (i) we used a real device, (ii) we generated a number of UI interactions and system events during the execution, and (iii) we collected 10 execution traces for each application (totaling 20 000 traces), in order to mitigate the occurrence of rare conditions and to stress several running options of the AUA.

More in detail, the executions were performed on a Google Nexus 5 with Android 4.4.4 (KitKat). The Nexus 5 is provided with a Qualcomm Snapdragon 800 chipset, a 32-bit processor quad core 2.3 GHz Krait 400 CPU, an Adreno 330 450 MHz GPU, and 2 GB of RAM. The used model had 16 GB of internal memory.

Concerning the UI interactions and system events, we used the monkey tool of the Android Debug Brigde (ADB [2]) version 1.0.32. Monkey generates pseudo-random streams of user events such as clicks, touches, or gestures; moreover, it can simulate a number of system-level events. Specifically, we configured Monkey to send 2000 random UI events in one minute and to stimulate the Android operating system with the following events (one every 5 s starting when the AUA is in foreground): (1) reception of SMS; (2) incoming call; (3) call being answered (to simulate a conversation); (4) call being rejected; (5) network speed changed to GPRS; (6) network speed changed to HSDPA; (7) battery status in charging; (8) battery status discharging; (9) battery status full; (10) battery status 50%; (11) battery status 0%; (12) boot completed. This set of events was selected because it represents an acceptable coverage of all possible events which an app can receive. Moreover, this list takes into account the events which most frequently trigger the payload in Android malware, according to [36, 37].

In order to collect the traces for an AUA, we built a script which interacts with ADB and the connected device and performs the following procedure:

1. copies the AUA into the storage device;
2. installs the AUA (using the `install` command of ADB);
3. gets the package name and the class (activity/service) of the AUA with the launcher intent (i.e., get the AUA entry point, needed for step 4);
4. starts the AUA (using the `am start` command of ADB);
5. gets the AUA process id (PID, needed for step 6);
6. starts system calls collection;
7. starts Monkey (using the `monkey` command of ADB), instructed to send UI and system events;
8. waits 60 s;
9. kills the AUA (using the PID collected before);
10. uninstalls the AUA (using the `uninstall` command of ADB);
11. deletes the AUA from the device.

Table 1: Statistics about the length $|t|$ of collected sequences forming our dataset $\mathcal{T}$.

Subset	Mean	1st qu.	Median	3rd qu.
Trusted	23 170	6425	15 920	31 820
Malware	12 020	2422	4536	11 160
All	17 600	3397	8198	23 390

Table 2: Percentage of ten most occurring system calls in our dataset, divided between traces collected for trusted (left) and malware (right) applications.

Call (trusted)	Perc.	Call (malware)	Perc.
`clock_gettime`	30.66	`clock_gettime`	28.78
`ioctl`	9.00	`ioctl`	8.85
`recvfrom`	8.67	`recvfrom`	8.85
`futex`	7.89	`epoll_wait`	7.50
`getuid32`	4.96	`getuid32`	6.64
`getpid`	4.84	`futex`	6.61
`epoll_wait`	4.78	`mprotect`	6.34
`mprotect`	4.67	`getpid`	5.64
`sendto`	4.60	`sendto`	2.74
`gettimeofday`	3.19	`cacheflush`	2.00

To collect system calls data (step 6 above) we used strace [4], a tool available on Linux systems. In particular, we used the command `strace -s PID` in order to hook the AUA process and intercept only its system calls.

The machine used to run the script was an Intel Core i5 desktop with 4 GB RAM, equipped with Linux Mint 15.

Tables 1 and 2 show salient information about the collected execution traces: the former shows the statistics about the length $|t|$ of collected sequences. It can be observed that the system calls sequences of trusted apps are, in general, much longer than those of malicious apps. This suggests that the behavior of trusted apps is much richer than the one of malicious apps, which is expected to be basically limited to the execution of the payload. From another point of view, malware apps could exhibit poorer variability behavior than trusted apps and hence recurring sequences, corresponding to the malware fingerprint, should be identifiable.

Table 2 shows the percentage of the ten most occurring system calls in our dataset, divided between traces collected for the trusted (left) and malware (right) applications. It can be seen that the simple occurrences of system calls is not enough to discriminate malicious from trusted apps. As a matter of fact, both malware and trusted applications exhibit the same group of most frequent system calls: `clock_gettime`, `ioctl`, `recvform` are the top three for both the samples.

4.3 Methodology and results

4.3.1 *Unseen execution trace*

For this scenario, we built the training set T by including 8 (out of 10) traces picked at random for each application in $\mathcal{T}$. The remaining 2 traces for each application were used for testing (i.e., $T' = \mathcal{T} \setminus T$). This way, several traces for each application and for each family were available for the training phase of our method. In other words, in this and the following scenario (Section 4.3.2), we used 80% of the available data for training and the remaining 20% for testing.

Table 3: Results on unseen execution traces.

Method	n	k	Accuracy	FNR	FPR
System calls	1	25	91.1	11.5	6.2
	1	50	92.0	10.0	6.0
	1	75	91.8	10.3	6.2
	2	250	96.1	3.8	4.1
	2	500	96.5	3.4	3.5
	2	750	96.3	4.0	3.4
	3	250	95.5	4.9	4.1
	3	500	96.2	4.4	3.1
	3	750	97.0	3.0	3.0
Permissions		25	56.9	97.7	0.2
		50	60.4	22.4	53.2
		100	69.8	44.6	18.9
		250	88.9	17.0	6.5
		500	90.4	16.3	4.3
		750	90.6	16.0	4.2

After the training phase, we applied our method to each trace in T' and measured the number of classification errors in terms of False Positive Rate (FPR)—i.e., traces of trusted applications classified as coming from malware applications—and False Negative Rate (FNR)—i.e., traces of malware applications classified as coming from trusted applications.

We experimented with different values for the length n of the calls subsequences and the number k of selected features—k' was always set to 2000. We varied n in 1–3 and k in 25–750, when possible—recall that k is the number of selected features among the $|C|^n$ available features, hence we tested only for the values of $k > |C|^n$, with a given n. In order to mitigate lucky and unlucky conditions in the random selection of the traces used for the training phase, we repeated the procedure 3 times for each combination of n, k by varying the composition of T and T'.

The parameters n and k represent a sort of cost of the detection method: the larger their values, the higher the amount of information available to the classifier and, hence, the effort needed to collect it. However, provided that some system mechanism was available to collect the system calls generated by each process, we think that no significant differences exist in an implementation of our method among different values for n and k which we experimented within this analysis.

Table 3 shows the results obtained with our method applied with several combinations of n and k values: for each combination, the table shows the average values of accuracy, FNR, and FPR across the 3 repetitions. The table also shows the results obtained with the baseline method (see Section 4.1). It emerges that the proposed method is largely better than the baseline, as we obtained a best-in-class accuracy of 97% with the former and 91% with the latter. It is important to notice that FNR is low for the highest values of n and k, and that such a value is balanced with FPR.

On the other hand, we note that 91% is a pretty high accuracy: this figure suggests that permissions indeed play an important role in Android malware detection. Yet, they are not enough to effectively discriminating malware, as the rate of false negative keeps high (16%). As a matter of fact, this value can be also explained by the common practice of "overpermissions" (malware writers tend to write a long list of permissions, including those which are not necessary to the app for hiding "suspect" permissions to users). Moreover, the permissions list is an indicator at a coarse grain for identifying malicious behavior, as it could happen that a malicious behavior requires the same permissions of a licit behavior.

Concerning the impact of n and k on the detection accuracy, it can be seen that, for both parameters, the higher the better. For n, this means that longer sequences of system calls better capture the application behavior, and are hence more suitable to constitute a fingerprint. On the other hand, k represents the number of sequences which the method deems relevant, i.e., the fingerprint size: results show that the longer the sequences, the larger the size needed to fully enclose the information represented by the sequences. However, we verified that larger values for k did not deliver improvements in the accuracy. It can be seen that increasing n the accuracy grows and it grows also faster with k.

With $n = 1$, our method just uses frequencies of system calls, rather than sequences: results of Table 3 show that this variant exhibits a lower detection rate, compared to the case in which sequences are considered. This finding suggests that the simple frequency of system calls can be a misleading feature: indeed, there are system calls which are more likely to be used in a malware rather than in a trusted app (e.g., `epoll_wait` and `cacheflush`, as reported in Table 2), but they do not allow to build a highly accurate classifier.

Concerning the execution times, the training phase of our method took, on the average for $n = 3$ and $k = 750$, 1183 s, of which 61 s for the feature selection procedure and 1122 s for training the SVM classifier: we found that these times grow roughly linearly with k' and k, respectively. The actual classification phase of a single trace took, on the average for $n = 3$ and $k = 750$, 18 ms: we found that this time is largely independent from n and grows roughly linearly with k. Finally, the trace processing (i.e., transforming a trace t in a feature vector $\mathbf{f}(t)$) took, on the average for $n = 3$, 135 ms: we found that this time grows roughly exponentially with n. We obtained these figures on a machine powered with a 6 core Intel Xeon E5-2440 (2.40 GHz) equipped with 32 GB of RAM: we remark that we used a single threaded prototype implementation of our method.

4.3.2 *Unseen application*

For this scenario, we built the training set T by including *all* the traces of 1600 (among 2000) applications picked at random and evenly divided in trusted and malicious.

Then, after the training phase, we applied our method to the traces of the remaining 400 applications and measured accuracy, FPR and FNR. This way, no traces for a given AUA were available for the training; however, traces for applications different from the AUA yet belonging to the same family could have been available for the training.

Table 4: Results on unseen applications, with $n = 3$.

k	Accuracy	FNR	FPR
250	94.1	7.0	4.8
500	94.7	6.4	4.3
750	94.9	6.1	4.2

Since the results of previous experiments show that the best effectiveness can be obtained with $n = 3$, we varied only k in 250–750, in order to investigate if the ideal fingerprint size changes when unseen applications are involved. We repeated the procedure 3 times for each value of k by varying the composition of T and T'.

Table 4 shows the results. The main finding is that our detection method is able to accurately detect ($\approx 95\%$) also unseen malware, provided that it belongs to a family for which some execution traces were available. This finding further validates that our method can build an effective malware fingerprint. Moreover, it supports our assumption that sequences of system calls capture the similarity of behavior among malware applications of known families.

Concerning the impact of k on the accuracy, it can be seen that 750 remains the value which delivers the best accuracy. Moreover, we found the upper limit of k value beyond which no significant accuracy improvement occurs, is lower than in the former scenario. We speculate that this depends on the fact that less families are represented in the training set, hence a slightly smaller fingerprint size is enough to fully exploit the expressiveness of sequences of $n = 3$ system calls.

4.3.3 *Unseen family*

For this scenario, we repeated the experiment for each malware family of our dataset.

In particular, for a given family, we built the training set T by including *all* the traces of *all* the malicious applications not belonging to that family and *all* the traces of *all* the trusted applications.

Then, after the training phase, we applied our method to all the traces of all the applications of the considered family and measured FNR. This way, no traces for a given AUA were available for the training; moreover, no traces for any application belonging to the same family of the AUA were available for the training. We executed this experimentation with $n = 3$ and $k = 750$.

Table 5 shows the results for the 10 families most represented in our dataset. We remark that, since we were interested in assessing our method ability in detecting malware belonging to unseen families, we did not tested it on trusted traces, which we instead used all for the training. It can be seen that FNR greatly varies among these families: it spans from a minimum of 3.5% to a maximum of 38.5%. Hence, there are some unseen families which could have been detected by our method (GinMaster and zHash); conversely, for other families the detection rate is remarkably lower. We think that this happens because some malware

Table 5: FNR on unseen families, with $n = 3$ and $k = 750$.

Family	FNR
DroidKungFu	31.8
GinMaster	3.5
BaseBridge	4.6
Geinimi	18.6
PJApps	23.7
GloDream	38.5
DroidDream	32.9
zHash	4.3
Bgserv	12.6
Kmin	27.4

family is more different, in terms of behavior, than others: hence, a fingerprint built without that family could not be effective enough to detect malware applications belonging to that family. We conjecture that a larger and more representative training set—as the one which could be available in a practical implementation of our approach—could address this limitation.

5. CONCLUDING REMARKS AND FUTURE WORK

We presented a method for detecting Android malware which is based on the analysis of system calls sequences. The underlying assumption is that a fingerprint of malware behavior can be built which consists of the relative frequencies of a limited number of system calls sequences. This assumption is supported by the fact that the typical evolution of Android malware consists in modifying existing malware, and hence behaviors are often common among different malicious apps. Moreover, capturing app behavior at such a fine grain allows our method to be resilient to known evasion techniques, such as code alteration at level of opcodes, control flow graph, API calls and third party libraries.

We used Machine Learning for building the fingerprint using a training set of execution traces. We assessed our method on 20 000 execution traces of 2000 apps and found that it is very effective, as it obtained a malware detection accuracy of 97%, which is high compared to previous works, most of which have been assessed on a much smaller dataset. Furthermore, our validation differs from the most discussed in literature, as it makes use of real devices rather than emulators or modified kernel, which makes the experiment more realistic. As future work, we plan to investigate the following concerns:

- Evaluate to which extent our method can withstand common evasions techniques [24, 25].
- Consider longer sequences (i.e., greater values of n), since this could allow to capture even better the malware behavior. Unfortunately, since the actual number of possible sequences grows exponentially with n, this also implies coping with a very large problem space.
- Improve the quality of the training data by labelling only those portions of the execution traces of malware applications which actually correspond to malicious behaviors.
- Extend our method to not only classify an entire execution trace as malicious or trusted, but also to specify exactly where, in the trace, there appears to occur the malicious behavior.

6. REFERENCES

[1] Google play. `https://play.google.com/store?hl=it`, last visit 24 November 2014.

[2] Android debug bridge. `http://developer.android.com/tools/help/adb.html`, last visit 25 November 2014.

[3] Google play unofficial python api. `https://github.com/egirault/googleplay-api`, last visit 25 November 2014.

[4] strace-linux man page. `http://linux.die.net/man/1/strace`, last visit 25 November 2014.

[5] Virustotal. `https://www.virustotal.com/`, last visit 25 November 2014.

[6] B. krebs. mobile malcoders pay to google play. `http://krebsonsecurity.com/2013/03/mobile-malcoders-pay-to-google-play/`, last visit 31 November 2014.

[7] Damballa labs. damballa threat report first half 2011. technical report, 2011. `https://www.damballa.com/downloads/r_pubs/Damballa_Threat_Report-First_Half_2011.pdf`, last visit 31 November 2014.

[8] Nqmobile. mobile malware up 163% in 2012, getting even smarter in 2013. `http://ir.nq.com/phoenix.zhtml?c=243152&p=irol-newsArticle&id=1806588`, last visit 31 November 2014.

[9] A. Armando, A. Merlo, and L. Verderame. An empirical evaluation of the android security framework. In *Security and Privacy Protection in Information Processing Systems IFIP Advances in Information and Communication Technology Volume 405, 2013, pp 176-189*, 2013.

[10] D. Arp, M. Spreitzenbarth, M. Huebner, H. Gascon, and K. Rieck. Drebin: Efficient and explainable detection of android malware in your pocket. In *Proceedings of 21th Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[11] A. Aswini and P. Vinod. Droid permission miner: Mining prominent permissions for android malware analysis. In *Proceedings of 5th International Conference on the Applications of Digital Information and Web Technologies (ICADIWT)*, 2014.

[12] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of 17th ACM Conference on Computer and Communications Security*, 2010.

[13] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. Detection of malicious web pages using system calls sequences. In *Proceedings of the 4th International Workshop on Security and Cognitive Informatics for Homeland Defense (SeCIHD 2014), in conjunction with the International Cross Domain Conference and Workshop (CD-ARES 2014) pp. 226-238*, 2014.

[14] G. Canfora, F. Mercaldo, and C. A. Visaggio. A classifier of malicious android applications. In *Proceedings of the 2nd International Workshop on Security of Mobile Applications, in conjunction with the International Conference on Availability, Reliability and Security*, 2013.

[15] F. Di Cerbo, A. Girardello, F. Michahelles, and S. Voronkova. Detection of malicious applications on android os. In *Proceedings of 4th international conference on Computational forensics*, 2011.

[16] M. Egele, T. Scholte, E. Kirda, and K. C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2), Feb 2012.

[17] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services, pages 281–294*, 2012.

[18] T. Isohara, K. Takemori, and A. Kubota. Kernel-based behavior analysis for android malware detection. In *Proceedings of Seventh International Conference on Computational Intelligence and Security, pp. 1011-1015*, 2011.

[19] Y.-s. Jeong, H.-t. Lee, S.-j. Cho, S. Han, and M. Park. A kernel-based monitoring approach for analyzing malicious behavior on android. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014.

[20] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. F. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th conference on USENIX security symposium, pp. 351-366*, 2009.

[21] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security, pp. 399-412*, 2010.

[22] X. Liu and J. Liu. A two-layered permission-based android malware detection scheme. In *Proceedings of 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, 2014.

[23] A. Porter Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of 18th ACM Conference on Computer and Communications Security*, 2011.

[24] D. M. T. M. M. Protsenko. Divide-and-conquer: Why android malware cannot be stopped. In *Proceedings of International Conference on Availability, Reliability and Security (ARES), pp. 30-39*, 2014.

[25] V. Rastogi, Y. Chen, and X. Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *Information Forensics and Security, IEEE Transactions on*, 9(1):99–108, Jan 2014.

[26] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of EuroSec*, 2013.

[27] A.-D. Schmidt, H.-G. Schmidt, J. Clausen, K. A. Yuksel, O. Kiraz, A. Camtepe, and S. Albayrak. Enhancing security of linux-based android devices. In *Proceedings of 15th International Linux Kongress*, 2008.

[28] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *Information Security Tech. Report*, 14(1), Feb 2009.

[29] G. Sheran. Android apps security, apress.

[30] M. Spreitzenbarth, F. Echtler, T. Schreck, F. C. Freling, and J. Hoffmann. Mobilesandbox: Looking deeper into android applications. In *28th International ACM Symposium on Applied Computing (SAC)*, 2013.

[31] A. Sung, P. Chavez, and S. Mukkamala. Accessminer: using system-centric models for malware protection. In *Proceedings of the 20th Annual Computer Security Applications Conference, pp. 326-334*, 2004.

[32] F. Tchakounté and P. Dayang. System calls analysis of malwares on android. In *International Journal of Science and Tecnology (IJST) Volume, 2 No. 9*, 2013.

[33] X. Wang, V. Jhi, S. Zhu, and P. Liu. Detecting software theft via system call based birthmarks. In *Proceedings of the Computer Security Applications Conference, pp. 149-158*, 2009.

[34] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Proceedings of 7th Asia Joint Conference on Information Security*, 2012.

[35] R. Xu, H. Saïdi, and R. Anderson. Aurasium: practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security symposium, pp. 27-27*, 2012.

[36] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of 33rd IEEE Symposium on Security and Privacy (Oakland 2012)*, 2012.

[37] Y. Zhou and X. Jiang. Android malware, springerbriefs in computer science, 2013.

[38] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012.

Tailoring Software Architecture Concepts and Process for Mobile Application Development

Felix Javier Acero Salazar, Marco Brambilla
Politecnico di Milano. Dipartimento di Elettronica, Informazione e Bioingegneria.
Piazza Leonardo da Vinci, 32. 20133 Milan, Italy
felixjavier.acero@mail.polimi.it, marco.brambilla@polimi.it

ABSTRACT

Enabled by the continuous improvement of the hardware and software in mobile devices, mobile applications have evolved into very complex pieces of software. Yet, such increase in complexity has not been paired by an increased awareness, among developers, of the important role that some software engineering processes play in managing such complexity.

In this paper we focus on the architectural design of mobile applications: we show how this aspect is still overlooked by mobile app developers; we present a high level process and several concepts that aim to guide developers in the creation of suitable architectures for their apps; and we describe the advantages of integrating architectural thinking within the mobile app development process.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.2.29 [**Software Engineering**]: Management

Keywords

Software architecture, mobile applications, architectural design, mobile software architecture.

1. INTRODUCTION

Mobile applications today are not any more simple content fruition interfaces that present data on the screen and capture user interactions. Instead, they also manage complicated business logic, enable mechanisms for accessing and synchronizing data stored in local and remote data sources, gather contextual information, enable communication paths to access external service providers and orchestrate the operation of a growing set of wearable devices and sensors to which they offer processing power and communication capabilities with the cloud and other networked systems.

Two important consequences of such increase in complexity are: *1)* mobile applications now need to be developed by larger developer teams; *2)* the processes used to develop and maintain these applications have become more relevant.

Given the rather narrow advice offered by platform developers like Apple and Google in this regard, a particular process in which we want to concentrate in this paper is the architectural design; for it is through it that mobile applications can achieve important quality attributes like modifiability, testability and reusability.

Following this line of thought, this paper aims to reinforce architectural thinking among mobile developers by mapping some architectural concepts on to the mobile world, and presenting a high level process that developers can follow to design suitable architectures for their apps.

DeMobile'15, August 31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3815-8/15/08...$15.00
http://dx.doi.org/10.1145/2804345.2804350

2. ARCHITECTURE: THE MISSING CONCERN OF MOBILE DEVELOPMENT

Over the past decades, software architecture has consolidated as an important sub-field of software engineering [7]. Its importance and value has resonated across practitioners to the point where it is hard to think of any reasonably complex software project that does not give special attention to the architecture of the system. In the mobile world, however, it seems that much remains to be done.

Mobile application developers typically obtain a good part of the guidance for how to implement their applications directly from platform developers. Apple and Google, for example, continuously publish design and programming guidelines that are meant to instruct developers in the creation of high quality apps. But when it comes to software architecture, the advice offered by some platform developers is rather narrow.

In the case of iOS, for example, Apple advises a particular flavor of the MVC pattern as the best way for programming iOS apps [10]. Following this vision, iOS applications are typically organized in Model-View-ViewController triplets that collaborate using the communication mechanisms that are endemic to the platform — i.e Delegates, Target/Actions, Outlets, Key-Value Observers and Segues. Using the MVC architectural pattern to structure an entire iOS application comes handy when developing simple applications; for it only requires a handful of classes to support all the application use cases. In contrast, following the same path when creating non-trivial applications may lead to known problems like the *Massive View Controllers* [1] issue.

While the *Massive View Controllers* issue can be considered as a phenomenon endemic to the iOS platform, it does serve to illustrate one of the challenges that software archi-

[1]http://www.objc.io/issue-1/lighter-view-controllers.html

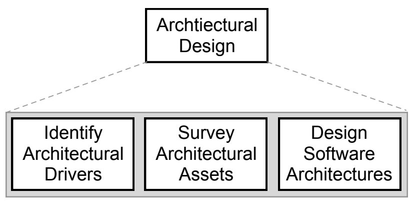

Archtiectural Design
Identify Architectural Drivers
Survey Architectural Assets
Design Software Architectures

Figure 1: Mobile Architecture Process

tecture faces in the mobile world: the **false intuition that the software architecture of mobile applications is a solved issue** and that following the guidelines of platform developers is enough to guarantee that a mobile application would have a good architectural design.

3. MOBILE SOFTWARE ARCHITECTURE PROCESS

To address this issue, our paper highlights the important role that the architectural design plays in the development of complex mobile applications. Our aim is not to propose a new architectural methodology; instead, we present **a high-level process that can guide application developers in the task of designing a suitable software architecture** for their mobile apps.

Several publications discuss the architectural design process [1]. We extract the most relevant stages of the architecture process for mobile applications, as shown in Figure 1: *Identify Architectural Drivers*, *Survey Architectural Assets* and *Design Software Architecture*. We will describe each of them in the following sections. Moreover, to provide a better idea of how each of the activities in the process maps onto a real life project, we will reflect over the experience that the software engineering team of Coursera[2] had while re-designing the architecture of their Android and iOS apps, based on two presentations given to the developer community [5] [14].

3.1 Identify Architectural Drivers

The first stage of the process is the identification of the main *architectural drivers* of the application. According to [1], the *architectural drivers* of an application are the collection of functional, quality and business requirements that shape the architecture of the application.

Though different mobile applications will likely have different *architectural drivers*, two quality attributes that may define the shape of the architecture of a large subset of mobile applications are: modifiability and testability.

Mobile applications today need to be designed for change. On the one hand, they need to adapt to the shifting business requirements that frequently imply the addition, update and deletion of some features of the application. On the other hand, since mobile applications frequently rely on third party libraries and Web APIs, and these are bound to change [4], having an architecture that allows for the easy replacement, deletion and addition of functionality, while managing the cost and scope of change, is an important goal.

Update cycles of mobile applications, on the other hand, are lengthier than those of web applications. Patching an application bug implies the release of an update for the application which needs to go through the revision and approval process of platform developers. Identifying and catching errors before an application is released is, therefore, an important concern of mobile software engineers. In consequence, having an application structure that enables easy testing for as many components as possible, is an increasingly important *architectural driver*.

In the case of Coursera, for example, the main quality attributes that shaped the structure of their architecture were: modifiability, reusability and testability. On the one hand, they needed to allow developer teams to work independently on different features of the application, which required a highly modular structure. On the other hand, they wanted to allow external developers to create their own features while leveraging the capabilities of the existing infrastructure, which in turn required that some of the core functionalities of the application had to be reusable and easily shared across developer teams. Finally, the software engineering team also wanted to enable more testing across the different components of the application to reduce the probability of shipping faulty applications.

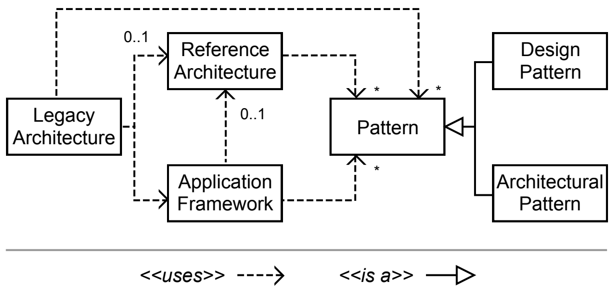


Figure 2: Architectural Assets Metamodel adapted from [3]

3.2 Survey Architectural Assets

Once application developers have identified the *architectural drivers* that may have an important influence on the final structure of the application, they should survey the field looking for inspiration that can help them solve the challenges imposed by these drivers. Since the final goal is to create a suitable software architecture for a mobile application, this survey should focus on *architectural assets*.

Architectural assets are defined by [3] as a set of reusable resources that can significantly help architects in their work since it reduces the number of things that the architect needs to be concerned with. *Architectural assets* are, therefore a good source of inspiration for mobile software engineers.

Although [3] provides an exhaustive list of the *architectural assets* that may be used by a software architect, in the following sections we present a subset that have special value in the mobile context. Figure 2 shows an adaptation of the metamodel presented by [3] that highlights the relationships that exist between some of these assets.

3.2.1 Architectural Patterns

An architectural pattern[3] is a coarse grained pattern that provides an abstract framework for a family of systems [13].

[2]https://www.coursera.org/

[3]also referred to as an architectural style in [7]

They help define the fundamental structure of the application [2] and specify a design vocabulary, constraints on how that vocabulary is used, and semantic assumptions about that vocabulary [7]. Examples of architectural patterns are: Layers, Pipe and Filter, Client/Server, N-Tier, Service Oriented and Message Bus [13, 7].

An architectural pattern of special relevance for rich mobile applications is the *Layers pattern*. The *Layers pattern* is described by [2], as an strategy to effectively divide the responsibilities of an application across the objects that compose it. Figure 3 (adapted from [13]), for example, shows a *reference model* of a rich mobile application organized according to the *Layers pattern*.

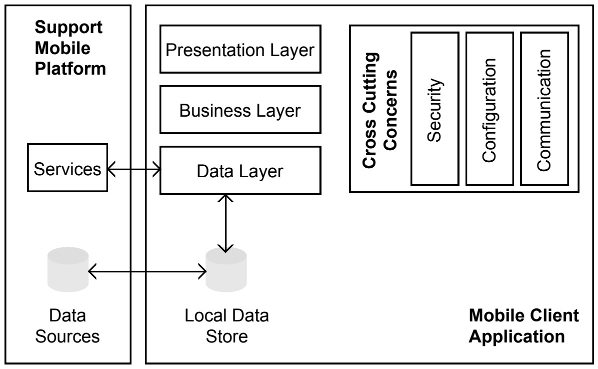


Figure 3: Layers Pattern adapted from [13]

Another pattern of notable importance for mobile applications is the MVC architectural pattern. However, as we discussed in Section 2, this pattern needs to be applied with caution as an architectural style, for it seems to be better suited for structuring thin or simple mobile apps, rather than rich or complex ones.

3.2.2 Reference Architectures

A reference architecture captures the basic building blocks and architectural design decisions of a system in certain application or technology domain [6]. They also represent reusable architecture knowledge in the form of generic artifacts, standards, design guidelines, architectural styles or domain vocabulary [6].

A particular reference architecture that has been widely discussed within the iOS developer community is VIPER [9]. VIPER is a reference software architecture, introduced by Mutual Mobile[4], with the aim of facilitating the design of software architectures for rich iOS apps. The main components of VIPER are: Views, Presenters, Interactors, Routers, Entities and Data Stores, and the relationships between them are shown in the Figure 4.

Although VIPER was introduced within the context of iOS development, nothing in its structure ties the architecture to a particular technology or mobile platform. In fact, when designing the architecture of their mobile apps, the software engineering team of Coursera used VIPER as the initial point from where the specific architectures for their iOS and Android applications were then extracted.

3.2.3 Application Frameworks

[4]http://www.mutualmobile.com/

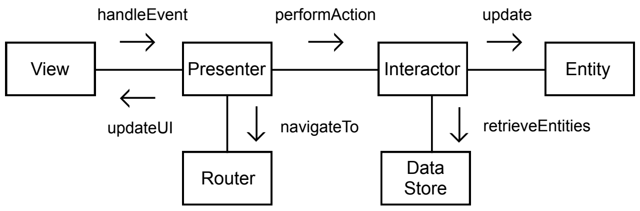


Figure 4: VIPER Architecture adapted from [9]

An application framework, as defined by [3], represents the partial implementation of a specific area of an application. In the mobile context, application frameworks play a fundamental role since they allow developers create sophisticated applications providing only the code that implements the differential aspects of their apps.

An application framework serves as an extensible skeleton that is customized by the methods defined by the user [11]. In consequence, when developing a mobile application, developers often find themselves writing code that is called by the methods within the frameworks. In iOS, for example, this will be the case of the code that developers write in UIViewController methods like viewDidLoad, viewWillAppear, viewDidAppear and similar. In Android, instead, the code written in Activity methods like onCreate, onResume or onPause would be equivalent examples.

In conclusion, since in the mobile context application frameworks are almost unavoidable, they are an important *architectural asset* and one that application developers need to get to know very well.

3.2.4 Legacy Architectures

Legacy architectures are an asset that can be defined as any kind of existing architecture. In the mobile world, this is an asset that is frequently available as software organizations typically publish different versions of their apps, and even have more than one application on the stores. The architectures used to develop all of these applications represent an important source of inspiration for mobile software architects, because all of them provide a privileged window to the structure of the system, help identify which strategies were effective, and hihglight some of the constraints imposed by the organization for granting coherency among its products.

3.3 Design the Software Architecture

The final step of the process, consists in the creation of a software architecture that is suitable for the mobile application. Inspiration for this task may come from different places, but according to [12], *theft, method and intuition* are among the most relevant. *Theft* refers to the inspiration that software architects may take from existing architectures, architectural patterns, reference models, application frameworks, and other sources (as we did in previous section); *Intuition* refers to the experience and skills of the software architect; and *method* refers to the systematic process by which the final architecture of an application is derived from its requirements.

Though several methods have been proposed in the research community, a particular method that aligns with the concepts that we have introduced so far is the *Attribute Driven Design* method, which decomposes a system or sys-

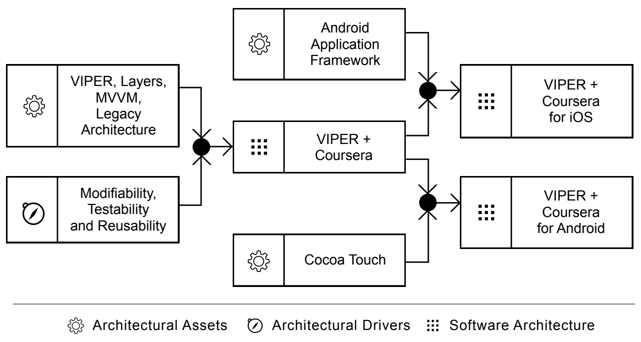

Android Application Framework
VIPER, Layers, MVVM, Legacy Architecture
VIPER + Coursera for iOS
VIPER + Coursera
Modifiability, Testability and Reusability
VIPER + Coursera for Android
Cocoa Touch
Architectural Assets
Architectural Drivers
Software Architecture

Figure 5: Coursera Mobile Software Architecture Process

tem element by applying architectural tactics and patterns that satisfy its driving requirements [1].

Applying this, or other method, in the mobile context however, should be done bearing in mind that the application frameworks used by the different mobile platforms may have an important influence over the shape of the final architecture. In consequence, a mobile software engineering team should first attempt to create a general architecture that satisfies the main *architectural drivers* at a *platform independent* level, and then apply some refinements over this general architecture to obtain the *platform specific* architecture for each of the target platforms. This is, in fact, the strategy used by the engineering team of Coursera, as discussed next.

4. THE COURSERA CASE

To see more clearly how the different steps of the process can provide a clearer view of the architectural design, we present the main drivers and assets that lead to the final architectural design of the Coursera application for Android and iOS.

In Figure 5 we can see how the VIPER *reference architecture* was a core asset that informed the final architectural design of the Coursera application. Other assets like *Layers pattern*, the *MVVM* architectural pattern [8], and the experience had with the previous architectural design of the application also help shape the final structure. On the other hand, requirements like allowing external developers to create features for the application, as well as allowing internal developer teams to work on different features independently were some of the architectural drivers that helped brew the final solution. Another aspect that is evident in Figure 5 is that the application frameworks used in Android and iOS also affected the architectural design process and led to the creation of platform specific architectures for each of the target platforms.

5. DISCUSSION AND CONCLUSIONS

In this paper we aimed to broaden the perspective of application developers regarding mobile software architectures, which become more relevant as mobile applications increase their complexity. Besides highlighting the importance of the architectural design, we presented a high-level process composed of three stages, aiming to guide developers in the creation of suitable architectures for their mobile apps.

The proposed process has three benefits: *1)* it is based on well known architectural processes [1, 3]; *2)* it maps the main concepts and knowledge of software architecture on to the mobile world; *3)* it advocates for a different view of mobile software architectures in which process and methodology are favored over predefined solutions. Finally, though the process represents a step towards the definition of architectural design practices for the mobile world, we think it is still too general when it comes to the actual design of the software architecture, and thus this work paves the way for further research to tailor some of the existing architectural methodologies for the mobile context.

6. REFERENCES

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison Wesley, second edition, 2003.

[2] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern Oriented Software Architecture - A System of Patterns.* John Wiley & Sons, Inc, 1996.

[3] Peter Eeles. Understanding architectural assets. *7th IEEE/IFIP Working Conference on Software Architecture, WICSA 2008*, pages 267–270, 2008.

[4] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Web API growing pains: Loosely coupled yet strongly tied. *Journal of Systems and Software*, 100:27–43, 2015.

[5] Mustafa Furniturewala. Building Modular iOS Apps. Using Swift, iOS 8 and MVVM to maintain a complex mobile app with a large team. https://realm.io/news/modular-ios-apps/, 2015.

[6] Matthias Galster. Software Reference Architectures : Related Architectural Concepts and Challenges. pages 5–8, 2015.

[7] David Garlan. Software architecture: a travelogue. *Proceedings of the on Future of Software Engineering*, pages 29–39, 2014.

[8] Raffaele Garofalo. *Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern.* Microsoft Press, Mar 2011.

[9] Jeff Gilbert and Conrad Stoll. Architecting iOS Apps with VIPER. http://www.objc.io/issue-13/viper.html, Jun 2014.

[10] Apple Inc. Cocoa Core Competencies. https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html, Sep 2013.

[11] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[12] Philippe Kruchen. Mommy, Where Do Software Architectures Come From? *1st International Workshop on Architectures for Software Systems, Seattle*, 1995.

[13] J.D. Meier, Alex Homer, David Hill, Jason Taylor, Lonnie Wall, Prashant Bansode, Akshay Bogawat, and Rob Boucher Jr. *Mobile Application Architecture Guide.* Microsoft, 2008.

[14] Yixin Zhu. Mobile Architecture and Android Design. https://youtu.be/iWf11tRRBB4, 2014.

Optimizing Energy of HTTP Requests in Android Applications

Ding Li and William G. J. Halfond
University of Southern California
Los Angeles, California, USA
{dingli,halfond}@usc.edu

ABSTRACT

Energy is important for mobile apps. Among all operations of mobile apps, making HTTP requests is one of the most energy consuming. However, there is not sufficient work in optimizing the energy consumption of HTTP requests in mobile apps. In our previous study, we found that making small HTTP requests was not energy efficient. Yet, we did not study how to optimize the energy of HTTP requests. In this paper, we make a preliminary study to bundle sequential HTTP requests with a proxy server. With our technique, we had a 50% energy saving for HTTP requests in two market Android apps. This result indicates that our technique is promising and we will build on the result in our future work.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Diagnostics

General Terms

Performance

Keywords

Energy optimization, HTTP requests, Mobile systems

1. INTRODUCTION

Energy is a critical resource for battery supported devices, such as smartphones and tablets. Improving the energy efficiency of mobile apps could improve the user satisfaction of the apps and potentially improve the revenue of developers. Recently, researchers have proposed many techniques to save energy for mobile apps. However, none of these approaches focus on the optimization of HTTP request energy consumption.

Unfortunately, overlooking HTTP requests is problematic. According to our previous study [8], making HTTP requests is the most expensive type of API call in mobile applications. On average, making HTTP requests can consume more than 32% of the total non-idle state energy of an app. Thus, researchers and developers can miss a significant area to optimize energy if they omit HTTP requests.

To study energy consumption of HTTP requests, we performed a preliminary study [7] and found that making small HTTP requests is not energy efficient. However, in that preliminary study, we did not propose a method to optimize HTTP energy.

In this paper, we introduce a preliminary study on how to optimize the energy consumption of HTTP requests in mobile apps. Our approach is to combine multiple HTTP requests that will definitely be made together. By doing this, our approach could reduce the number of HTTP requests and reduce the overhead of initiating and sending each individual HTTP request. In our evaluation, we found that there was an up to 50% reduction in the HTTP energy consumption in two market Android apps when using our approach.

The structure of this paper is as follows. In Section 2, we introduce some background information about making HTTP requests and our previous results. In Section 3, we introduce our ideas of how to optimize the HTTP request energy. In Section 4, we report the results of our evaluation. Finally, in Section 5, we discuss related work and conclude in Section 6.

2. BACKGROUND AND MOTIVATION

In our previous empirical study [8], we found that making HTTP requests is one of the most energy consuming operations in Android apps. In the empirical study, we measured the energy consumption of different packages of APIs in 405 Android market apps. We found that, on average, making HTTP requests could represent 32% of the total non-idle state energy consumption of mobile apps. For certain apps, this percentage could be even higher than 60%. Compared with other operations, making HTTP requests consumes significantly more energy.

Another one of our previous studies [7] further showed that making small HTTP requests is not energy efficient. In that study, we found that downloading one byte of data consumed the same amount of energy as downloading 1,024 bytes of data through HTTP requests. Furthermore, we found that downloading 10,000 bytes of data only consumed twice the amount of energy as downloading 1,000 bytes of data. In this case, making small HTTP requests consumes more energy per each byte transmitted through the network.

This inefficiency is due to the protocol of sending HTTP requests. Making an HTTP request needs three steps: es-

DeMobile'15, August 31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3815-8/15/08...$15.00
http://dx.doi.org/10.1145/2804345.2804351

tablishing the connection, transmitting data, and closing the connection. In the steps of establishing the connection and closing the connection, the client side needs to have a 3-way or 4-way handshake protocol, which spend energy in transmitting practically empty packets. In the step of transmitting data, energy overhead will be introduced by the headers of the HTTP request and its lower-level network protocols, such as TCP and IP. Thus, most energy of a small HTTP request will be consumed by the handshake-protocol and headers.

3. APPROACH

Our approach focuses on optimizing the HTTP energy consumption of mobile apps with sequential HTTP requests. By sequential HTTP requests, we mean the HTTP requests that are always made together, in sequence, despite the user input or execution condition of the program. One example of sequential HTTP requests is in Program 1, where the HTTP requests at line 6, line 10, and line 14 are sequential HTTP requests. Since these HTTP requests will always be sent in sequence, their tasks can be accomplished in one HTTP request.

```
 1 public void print_html()
 2 {
 3     URL url1, url2, url3;
 4     URLConnection urlConnection1,urlConnection2,
           urlConnection3;
 5     //query current weather
 6     url1 = new URL("http://weather");
 7     urlConnection1 = url1.openConnection();
 8     ParseStream(urlConnection1.getInputStream());
 9     //query weather forcast
10     url2 = new URL("http://daily");
11     urlConnection2 = url2.openConnection();
12     ParseStream(urlConnection2.getInputStream());
13     //query location info
14     url3 = new URL("http://location");
15     urlConnection3 = url3.openConnection();
16     ParseStream(urlConnection3.getInputStream());
17 }
```

Program 1: Example of sequential HTTP requests

To optimize the energy consumption of sequential HTTP requests, we propose an approach that combines sequential HTTP requests into a larger HTTP request. This approach can reduce the number of HTTP requests and increase the size of data transmitted by a single HTTP request. Our basic idea is to redirect the original HTTP requests to a proxy server that combines the sequential HTTP requests into a single one request. The work-flow of our approach is shown in Figure 1, where the dashed-line boxes represent the code from the original app and solid-line boxes represent the components that are developed in our approach. In our workflow, we first use code rewriting techniques to replace the Android HTTP APIs with the Agent HTTP APIs in an Android app. Then the Agent HTTP APIs will redirect the HTTP requests to the Proxy, which is a process that intercepts the HTTP requests from the client to the server. It contains two components, the Bundling Calculator and the Redirector.

When the client starts to send an HTTP request, it sends the request through the Agent HTTP APIs. Then, the Agent HTTP APIs redirect the request to the Proxy. When the Proxy gets the HTTP request, it passes the request to the Bundling Calculator, which calculates if there are any other HTTP requests should be bundled. Its decision is based on the rules provided by developers of the app. After that, the Bundling Calculator uses the Redirector to query for the data of all of the HTTP requests that need to be bundled from the server. When the responses are retrieved, the Bundling Calculator bundles the responses for all HTTP requests in one package and replies with it to the Agent HTTP APIs. Then, when the bundled responses return, the Agent HTTP APIs unpack the bundled responses and recover the response for each request. These responses are then store in a local cache or returned for the current HTTP request. Finally, when other HTTP requests in the same set of sequential HTTP requests are invoked, the responses will be returned directly from the local storage.

3.1 Example

We use the example in Program 1 to explain our approach. In Program 1, we first replace the HTTP APIs at line 8, line 12, and line 16 with our Agent HTTP APIs. When the HTTP request at line 8 is made, it is sent to the Proxy. Then the Proxy finds the incoming request is the first one of a set of sequential HTTP requests, which contains three HTTP requests of the link *weather* at line 6, *forecast* at line 10, and *location* at line 14. Such a decision is based on rules provided by the developers manually with their domain specific knowledge. Then the Proxy queries for the response to *weather*, *forecast*, and *location* from the server and returns them all in a single packet to the client.

When the Proxy sends back the packed response, our Agent HTTP APIs will capture this response at line 8 in Program 1. The Agent HTTP API first unpacks the response from the Proxy, retrieves the response for *forecast* and *location* from the packed response, and stores them in the local cache. Then it retrieves the response for *weather*, which is the target URL of the HTTP request at line 8, and returns the response as an InputStream, which is the same data structure as the original Android HTTP request.

After the Agent HTTP API at line 8 returns, the code makes the HTTP request to *forecast* at line 12 with the Agent HTTP APIs. At line 12, the Agent HTTP API checks the local cache and finds that the response is already stored, so it returns the response of *forecast* directly. This is the same for the case for *location*.

In this case, Program 1 only makes one HTTP request to the Proxy at line 8, which retrieves the responses for *weather*, *forecast*, and *location* together. This compared to the original version of Program 1, in which there are three HTTP requests made to retrieve the responses for *weather*, *forecast*, and *location*.

3.2 Code Rewriting

Our approach first needs to replace the original Android HTTP APIs with the Agent HTTP APIs. In this process, our approach parses the bytecode of each method in the original app and detects the signature of each invocation instruction. If there is an invocation instruction invoking an Android HTTP API, our approach redirects the invocation instruction to the corresponding Agent Android APIs.

3.3 The Agent HTTP APIs

The Agent HTTP APIs are static methods that replace the original HTTP APIs in Android SDK, such as *URLConnection.getInputStream*. The parameters of the Agent

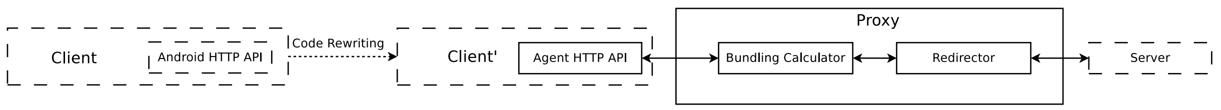


Figure 1: The work flow of our approach

HTTP APIs are the target object and the parameters of the original HTTP API. The return values of the Agent HTTP APIs are the same as the original API.

The Agent APIs have three main functions. First, check if the response for the current HTTP request is stored in the local cache, and if so, return it directly. Second, redirect the HTTP request to the Proxy instead of the server if the current HTTP request is not stored in the local cache. Third, unpack the bundled responses for a set of sequential HTTP requests, store the responses for them in the local cache, and return the response for the current HTTP request.

3.4 The Proxy

The Proxy is the process on the server side that intercepts the incoming HTTP requests between the client and the server. The first responsibility of the Proxy is to check if the incoming HTTP request is the first request in a sequence of HTTP requests and its second responsibility is to bundle the sequential HTTP requests. The Proxy accepts HTTP requests from the client and returns the bundled responses for those requests. The Proxy is normally deployed on the same machine or in the same network domain of the server. Thus, it can have a very high-speed connection to the server and will not significantly increase the latency between the server and the client. We believe this requirement is realistic because the user of our approach, who are the owners or developers of the app under optimization, generally have control of the app's server.

The first task is accomplished by the Redirector, which accepts the incoming HTTP request, retrieves the URLs of the corresponding sequential requests and, sends those requests to the server to get the responses.

The second task is accomplished by the Bundling Calculator, which accepts the URL and parameters of an incoming HTTP request and determines which requests should be bundled. This decision is made based on rules provided by the app developers with their domain specific knowledge about the app. For example, in Program 1, the developers can provide the rule "bundle the requests to *weather*, *forecast*, and *location* together" to the Bundling Calculator. Then, when the request of *weather* comes, the Bundling Calculator would query this rule and then bundle the response of *weather*, *forecast*, and *location* in one package.

3.5 The Communication Protocol

The Agent HTTP APIs and the Proxy use a specific protocol for communication. In this protocol, responses for several HTTP requests can be returned in a single packet in JSON format. When the Proxy returns data to the Agent HTTP APIs, it contains three types of information: first, the number of responses; second, the current request and its response; and third, all of the bundled requests and their responses.

When the Agent HTTP APIs get the response, they process the data with the following steps. First, they retrieve the response for the current request, this response will be used as the return value. Second, they iterate over all of the bundled requests, and retrieve their responses. Third, they cache the responses of all of the bundled requests in a local cache.

4. EVALUATION

In our evaluation, we answer the research question: how much HTTP energy could be saved by bundling the sequential HTTP requests as one HTTP request?

4.1 Test Apps

To answer our research question, we found 2 market Android apps from the Google Play Market that contain sequential HTTP requests. These two apps are **bob's weather** and **LIRR Schedule**. These apps have 22,517 and 4,408 lines of Java bytecodes, respectively.

4.2 Implementation

In the implementation, we used dex2jar and apktool to decompile the Dalvik bytecode of Android apps to Java bytecode. Then we used the BCEL library to replace the APIs that make HTTP requests with our Agent HTTP APIs. We used Node.js to implement the proxy server.

4.3 Evaluation Protocol

For our evaluation, we could not obtain the server side code of the apps since they were not open source apps. Thus, to evaluate our technique, we mimicked the behavior of the original server with a Node.js based mock-server. In our mock-server, we recorded and stored the responses of all HTTP requests that could be made by our two apps. When there was any incoming HTTP request, the mock-server simply replied with stored data according to the URLs and parameters of the requests. The mock-server and the proxy were all deployed on the Amazon EC2 cloud platform.

To evaluate the energy savings, we used our previous technique, vLens [9], to measure the HTTP energy consumption of the two apps. We compared the HTTP energy of the unoptimized version, which accessed the mock-server directly, with the HTTP energy of the optimized version, which accessed the proxy server, and reported the energy savings of our approach

4.4 Result

In our experiment, we found that there were energy savings of 50% for both apps for making HTTP requests. We believe this result is significant because as reported in our previous study, on average, making HTTP requests can consume 30% of the total non-idle state energy [8], we expect that bundling sequential HTTP requests could have a 10-15% reduction in the energy at the application level.

5. RELATED WORK

A large group of current studies on optimizing energy for mobile devices focus on detecting resource leakage of sensors (e.g., [13, 1]). Pathak and colleagues [15] proposed a technique to detect if an app fails to release a wake-lock of in a smartphone app.

Another group of energy optimization techniques focus on optimizing the display energy of mobile apps. Our previous work, Nyx [10], optimizes the energy consumption of mobile web apps by automatically transforming the color scheme of the apps. Mian and colleagues proposed dLens [17] to detect the display energy hotspots. Dong and colleagues also proposed a tool, Chameleon [3], to change colors of mobile web apps manually.

Besides detecting sensor resource leakage and display energy optimization, another group of techniques optimizes the energy consumption of mobile apps. Our previous work, EDTSO [11], optimizes the energy consumption of in-situ test suits by mapping the test suit optimization problems to an integer linear programming problem. Bruce and colleagues [2] used Genetic Improvement technique to optimize the energy consumption of MiniSat programs. Manotas and colleagues proposed a framework, SEEDS [14], to automatically make energy optimization decisions.

Although all approaches mentioned in the above three paragraphs achieved significant effect on energy optimization of mobile apps, none of them focus on optimizing the energy consumption of HTTP requests. As we discussed in Section 2, making HTTP requests is the most energy consuming operation in mobile apps.

Besides energy optimization, many techniques are also proposed to measure and estimate the energy consumption of mobile apps. In our previous work, we proposed two tools, eLens [5] and vLens [9], to estimate and measure the energy consumption of mobile apps at the source line level. Hindle [6] proposed a frame work, Green Mining, to measure energy consumption of mobile apps and related it with app code changes. eProf [15] models energy with a state machine. All of these approaches only measure the energy consumption of mobile apps, but do not do any optimization.

The last group of our related work is empirical studies. Our previous studies [7, 8] investigated how energy is generally consumed in mobile apps. Gui and colleagues performed an empirical study [4] about the hidden costs of mobile ads. Their hidden costs also included energy consumption. Linares-Vasquez and colleagues [12] conducted an empirical study on analyzing API methods and mining API usage patterns in Android apps. Li and colleagues [11] studied the energy consumption of different storage systems. Sahin and colleagues [16] proposed an empirical study about energy impact of code ossification.

6. CONCLUSION AND FUTURE WORK

Making HTTP requests is one of the most energy consuming operation in mobile apps. In this paper, we performed a preliminary study on how to optimize the energy consumption of HTTP requests in Android apps. Our approach is to bundle sequential HTTP requests into a single request. By doing so, we achieved 50% energy savings in HTTP requests in two Android market apps. Based on this result, we conclude that our technique is promising.

In our feature work, we will develop new techniques to automatically generate the bundling rules in the proxy and evaluate our technique with more market apps. Further, in this paper, our technique only optimizes the sequential HTTP requests in a single thread. In our future work, we will also optimize HTTP requests across different threads.

7. ACKNOWLEDGMENTS

This work was supported by NSF grant CCF-1321141.

8. REFERENCES

[1] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *FSE*, 2014.

[2] B. R. Bruce, J. Petke, and M. Harman. Reducing energy consumption using genetic improvement. In *17th Annual Conference on Genetic and Evolutionary Computation. ACM*, 2015.

[3] M. Dong and L. Zhong. Chameleon: A Color-Adaptive Web Browser for Mobile OLED Displays. *IEEE Transactions on Mobile Computing*, 2012.

[4] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *ICSE*, 2015.

[5] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating Mobile Application Energy Consumption Using Program Analysis. In *ICSE*, 2013.

[6] A. Hindle. Green mining: A methodology of relating software change to power consumption. In *MSR*, pages 78–87. IEEE Press, 2012.

[7] D. Li and W. G. Halfond. An Investigation Into Energy-Saving Programming Practices for Android Smartphone App Development. In *GREENS*, 2014.

[8] D. Li, S. Hao, J. Gui, and H. William. An Empirical Study of the Energy Consumption of Android Applications. In *ICSME*. IEEE, 2014.

[9] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating Source Line Level Energy Information for Android Applications. In *ISSTA*, 2013.

[10] D. Li, H. Tran, Angelica, and G. J. Halfond, William. Making Web Applications More Energy Efficient for OLED Smartphones. In *ICSE*, 2014.

[11] J. Li, A. Badam, R. Chandra, S. Swanson, B. L. Worthington, and Q. Zhang. On the energy overhead of mobile storage systems. In *FAST*, pages 105–118, 2014.

[12] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *MSR*, 2014.

[13] Y. Liu, C. Xu, and S. Cheung. Where has my battery gone? finding sensor related energy black holes in smartphone applications. In *PerCom*, 2013.

[14] I. Manotas, L. Pollock, and J. Clause. Seeds: A software engineer's energy-optimization decision support framework. In *ICSE*, 2014.

[15] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *MobiSys*, 2012.

[16] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause. How does code obfuscation impact energy usage? In *ICSME*, 2014.

[17] M. Wan, Y. Jin, D. Li, and W. G. J. Halfond. Detecting display energy hotspots in android apps. In *ICST*, April 2015.

Perspectives on Static Analysis of Mobile Apps (Invited Talk) [*]

Marco Autili[1], Ivano Malavolta[2], Alexander Perucci[1], Gian Luca Scoccia[2]
[1]University of L'Aquila, Department of Information Engineering, Computer Science and Mathematics (Italy)
[2]Gran Sasso Science Institute, L'Aquila (Italy)
marco.autili@univaq.it, alexander.perucci@graduate.univaq.it,
{ivano.malavolta | gianluca.scoccia}@gssi.infn.it

ABSTRACT

The use and development of mobile apps is growing at a tremendous rate in the last years. Even if this growth is making the mobile apps market very attractive for software developers, it is also continuously presenting new challenges. Indeed, mobile platforms are rapidly and continuously changing, with the addition of diverse capabilities like the support for new sensors, APIs, programming abstractions, etc. In this respect, a number of static analysis methods and techniques have been proposed in research as a powerful instrument for developing more qualitative mobile apps.In this invited talk we report on the results of a preliminary survey we conducted on *static analysis methods and techniques of mobile apps*.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

Keywords

Mobile applications, Static analysis, Systematic study

1. INTRODUCTION

Motivation – Programming languages and tools for developing mobile apps are platform-specific (e.g., Java code for Android apps, and Objective-C code for Apple iOS apps), and present many challenges that may hamper the success of a mobile app as a whole. Indeed, recently, an empirical study indicates a strong need of mobile app developers for better analysis and testing support, with a focus on important features like mobility, location services, sensors, as well as different gestures and inputs [3]. Under this perspective, *static program analysis* [4] can be an effective and viable instrument to predict and evaluate (precise or approximated) quantitative and qualitative properties related to the run-time behaviour of a mobile app without actually executing it. Static analysis of mobile apps can be a valuable instrument for both app developers an app store moderators (e.g., Google, Apple) because it helps in creating products with better quality in a world where a low-quality release can have devastating consequences [3]. In this respect, a number of static analysis approaches have been proposed in the literature, each of them with specific peculiarities, features, and capabilities.

Contribution – In this invited talk we report on the results of a preliminary survey we conducted on methods and techniques for static analysis which estimate specific properties and features of mobile apps. The *main outcomes* of this study are: (i) a reusable comparison framework for understanding, classifying, and comparing techniques for static analysis of mobile apps, and (ii) an initial overview of the current state of the art about existing methods and techniques for static analysis of mobile apps.

[*]This work is partially supported by the MIUR, prot. 2012E47TM2, Prin project IDEAS.

DeMobile'15, August 31, 2015, Bergamo, Italy
ACM. 978-1-4503-3815-8/15/08
http://dx.doi.org/10.1145/2804345.2804352

2. STUDY DESIGN

We designed this study by borrowing some principles from the well-known guidelines for performing systematic mapping studies in software engineering [5]. This kind of empirical strategy is extremely powerful in structuring a given research area (static analysis for mobile apps in our case) by collecting and analysing existing work in it, while minimizing as much as possible the chances of bias [5]. To the best of our knowledge, this study presents the first systematic investigation into static analysis for mobile apps.

Research questions. Goal of our study is to *identify and classify the characteristics and evaluation quality of methods and techniques for static analysis of mobile apps*. This goal can be refined into the following research questions:

RQ1. What are the existing methods and techniques for static analysis of mobile apps?

RQ2. What are the characteristics of existing methods and techniques for static analysis of mobile apps?

RQ3. What is the quality of the evaluation performed on existing methods and techniques for static analysis of mobile apps?

Studies search and selection. As the research area of static analysis for mobile apps is very recent (the concept of mobile app exists only since 2007) and given the preliminary nature of this study, we decided to focus exclusively on high-quality publications in top-level scientific venues in the software engineering area. Based on this, we performed a manual search on the top-level software engineering conferencesand international journals. The time span of our search is from January 2007 to March 2015, summing up to 6541 potentially relevant studies. Once identified the data sources, we considered all the selected studies and filtered them according to a set of well-defined inclusion and exclusion criteria. As recommended in the guidelines for performing systematic literature reviews [5], the selection criteria of this study have been decided upfront, so to reduce the likelihood of bias. In order to reduce bias, two researchers performed the studies selection independently. At the end of this stage we obtained the 9 primary studies.

Data extraction. The first goal of this stage is to create a comparison framework that fits well with the studies under analysis. We followed a systematic process called *keywording* [5] for defining the various parameters of our comparison framework. Once the comparison framework has been setup, we considered all primary studies and we populated the comparison framework with the extracted data. In order to mitigate the presence of biases, (i) two researchers extracted the data from all the primary studies independently, and (ii) we performed a sensitivity analysis to analyse whether the results are consistent independently from the researcher performing the analysis.

Data synthesis. We performed content analysis, mainly for categorizing and coding approaches under broad thematic categories. Then, we performed narrative synthesis for explaining in details and interpreting the findings coming from the content analysis.

3. RESULTS

In the following we report a selection of the main results of our survey and briefly discuss them. To allow easy replication and verification of our study, we make publicly available the replication package containing all the extracted data of the study[1].

Analysis Goal. Most of the primary studies focus on energy consumption. The second most common goal is to improve the security of mobile apps with techniques aimed at detecting malware and leaks of sensitive information. Other goals are (i) the detection of bugs that undermine performances, (ii) the reduction of memory consumption, and (iii) the detection of bugs.

Hybrid. In two cases the proposed static analyses are complemented with some kind of dynamic analysis. More specifically, in the first case the authors "*execute the instrumented code on the emulator and record the system calls invoked for each event trace* ", whereas in the other case the approach is complemented with a "*runtime measurement*".

Analysis Presteps. Three are the primary studies requiring preliminary steps before the static analysis. In the first one "*the Workload Generator is responsible for converting the user-level actions, for which the developer wants an estimation, to the path information used by the Analyzer and Source Code Annotator*"; in the second one "*preprocessing of application can be divided into three steps: (i) EFG [Event Flow Graph] extraction (ii) Event trace generation (iii) Extraction of system calls sequence for each event trace*"; finally, in the third one a preliminary run-time measurement is performed before the static analysis.

Abstraction Level. Static analyses may operate at different levels of abstraction: bytecode, source code, and intermediate model. Most of proposed approaches work on the bytecode or source code level of the app. Only one approach runs on an intermediate model of the app, which specifies the signature of malware families via a Datalog-based language. Interestingly, in two other case the bytecode of the app is used in conjunction with its source code in order to produce source code annotations for the developer.

Platform. The majority of the approaches are specific to the Android platform. A possible interpretation of this trend may be due to the open-source nature of the Android platform. Also, Android app binaries can be straightforwardly disassembled with off-the-shelf software libraries and their internal structure and contained static resources are easily analyzable in an automatic manner. Interestingly, the approaches presented in three cases are generic and applicable to a variety of platforms (e.g., Android, iOS, Windows, Blackberry), however their tool is implemented for the Android platform only.

[1]`http://cs.gssi.infn.it/demobile_2015`

4. RELATED WORK

In [2] a survey about static analysis and model checking approaches for searching patterns and vulnerabilities within a software system is proposed. Peculiarity of this research is the comparison of static analysis algorithms against mathematical logic languages for model checking. The authors of [6] conducted a survey about static analysis for identifying security issues and vulnerabilities in software systems in general (not specific to mobile apps). For each type of security vulnerability the authors present both relevant studies and the implementation details of the used static analysis algorithms. A systematic mapping study has been conducted in[1] for classifying and analysing approaches that combine different static and dynamic quality assurance technique. The study included a discussion about reported effects, characteristics, and constraints of the various existing techniques. In conclusion, even if there are studies about static analysis and some aspects of mobile apps, none of them is actually focussing on the analysis of mobile apps. Since the need of analysis of apps has been raised in some recent works (e.g., in [3]), our study falls exactly in this area of research by providing an initial overview of existing approaches for static analysis of mobile apps.

5. FUTURE WORK

As future work, we are working on the design and conduction of a full-fledged systematic mapping study about static analysis of mobile apps, with the chief aim, among the others, of performing a more systematic search and selection activity and a more thorough analysis of the obtained data. Also, according to the research gaps we will likely identify in our systematic map, we will focus on a specific research challenge in the area and we will propose methods and techniques for addressing it.

6. REFERENCES

[1] F. Elberzhager, J. Münch, and V. T. N. Nha. A systematic mapping study on the combination of static and dynamic quality assurance techniques. *Information and Software Technology*, 54(1):1–15, 2012.

[2] I. Garcıa-Ferreira, C. Laorden, I. Santos, and P. G. Bringas. A survey on static analysis and model checking. In *International Joint Conference SOCO'14-CISIS'14-ICEUTE'14: Bilbao, Spain, June 25th-27th, 2014, Proceedings*, volume 299, page 443. Springer, 2014.

[3] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *Empirical Software Engineering and Measurement, 2013*, pages 15–24, 2013.

[4] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2015.

[5] K. Petersen, S. Vakkalanka, and L. Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, 2015.

[6] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2):265–288, 2007.

A Mobile Application for Geographical Data Gathering and Validation in Fieldwork (Invited Talk)

Karine Reis Ferreira,
Lúbia Vinhas,
National Institute for Space Research (INPE), Brazil
{karine, lubia}@dpi.inpe.br

Cláudio Henrique Bogossian,
André F. Araújo de Carvalho
Foundation of Science, Technology and Space Applications (FUNCATE), Brazil
{claudio.bogossian, andre.carvalho}@funcate.org.br

ABSTRACT

Mobile devices, such as smartphones and tablets, are useful tools for *in situ* collecting information about spatial locations. In this paper, we describe the architecture of a mobile application for geographical data gathering and validation in fieldwork. This application is being developed based on well-established standards in order to assure spatial data interoperability between existing Spatial Data Infrastructures (SDI) and mobile systems.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architecture – *domain-specific architectures.*

Keywords

Mobile application, spatial data, interoperability

1. INTRODUCTION

The recent advancements of GPS, wireless communication network and portable technologies have motivated the use of mobile devices for *in situ* gathering information about spatial locations and validating geographical data [1][2]. Tsou [1] defines the term *mobile Geographical Information System (mobile GIS)* to refer to an integrated technological framework for accessing geospatial data and location-based services through mobile devices, such as smartphones and tablets. He argues that there are two major application areas of mobile GIS, *field-based* GIS and *location-based* services. This work focuses on mobile field-based GIS, that is, mobile systems for geographical data collection and validation in the field.

Two examples of projects that need mobile field-based GIS are PRODES (Monitoring of Brazilian Amazon Rainforest) and DETER (Real Time Deforestation Detection System), developed by Brazilian Institute for Space Research (INPE) [3]. PRODES has been yearly monitoring deforestation since 1988 whereas DETER has been producing near real-time deforestation and forest degradation alerts for more than 5 million Km^2 in the Brazilian Legal Amazon. Specialists of these two projects require mobile systems to collect extra information about deforested regions (e.g. photos) and validate them in the field, including places where there is limited or any network connectivity available.

DeMobile'15, August 31, 2015, Bergamo, Italy
ACM. 978-1-4503-3815-8/15/08
http://dx.doi.org/10.1145/2804345.2804354

Therefore, an essential feature of geographical data collection and validation mobile systems is the capability of working *offline*. To meet the demands of these two projects, this paper presents an ongoing work on designing and implementing a mobile application for geographical data gathering and validation in fieldwork. Section 2 presents its architecture and section 3, its implementation issues.

2. ARCHITECTURE

TerraMobile App is the name of the mobile application for geographical data gathering and validation in fieldwork developed in this work. Figure 1 presents its general architecture.

TerraMobile App has two modules for accessing geographical data, "*Online Data Access*" and "*Offline Data Access*". The "Online Data Access" module accesses geographical data from Spatial Data Infrastructures (SDI) through two kinds of well-known Open Geospatial Consortium (OGC) web services, Web Map Server (WMS) and Web Feature Server (WFS) [4] [5]. This module only works online and will be used when there is network connectivity available in the fieldwork.

SDI is a sharing platform that facilitates the access and integration of multi-source spatial data in a holistic framework with a number of technological components including policies and standards [6]. Nowadays, many data providers throughout the world have created their own SDIs, organizing and disseminating their geospatial data sets and metadata on the Internet via OGC web services. Accessing spatial data sets from distinct SDIs can improve the geographical data collection and validation task.

The "Offline Data Access" module works offline and is responsible for accessing geographical data in the mobile storage memory. We propose to store them in OGC Geopackage files [7].

The Geopackage specification defines a SQL database schema designed for the SQLite software library. This schema contains a set of pre-defined tables with integrity assertions, format limitations and content constraints to store spatial data sets and their metadata. GeoPackage files are platform-independent SQLite database files that contain vector and tiled raster data sets as well as their metadata. They are interoperable across different platforms, including personal computing environments and mobile devices.

To prepare GeoPackage files to be used in the mobile application, we are developing a plugin, called **TerraMobile plugin**, for the Geographical Information System (GIS) TerraView. TerraView is a general-purpose GIS developed using the TerraLib GIS library [8]. TerraView supports the development of plugin to enhance its functionalities.

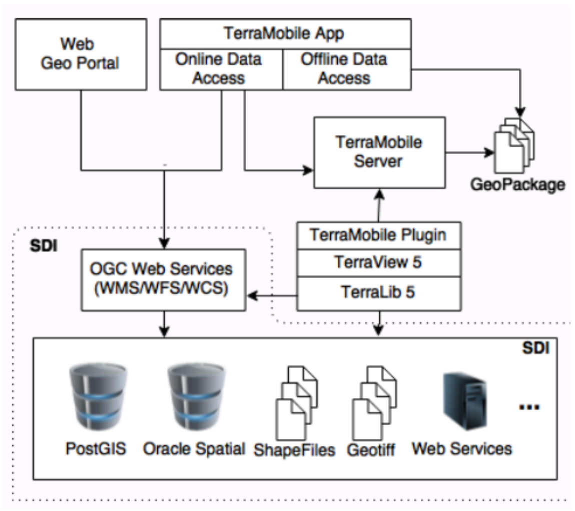


Figure 1 – TerraMobile architecture

TerraMobile plugin allows users to delimit an interesting area, access spatiotemporal data sets from different kinds of data sources and generate Geopackage files from these data sets. The generated files will be available through the **TerraMobile server**. Besides that, users can define the forms for acquiring data on fieldwork and synchronize the gathering data with the SDI using the TerraMobile plugin.

3. IMPLEMENTATION

TerraMobile App is being implemented for Android operation systems. We are using Android SDK (Software Development Kit) and Java language. Figure 2 shows the TerraMobile App.

In most cases, we are using native Android graphic components. However, we are also using some third party libraries to show components like maps and access geospatial data. We are using the following technologies:

- **Android SDK Api 15+ (4.0.3 ICS)**: the lowest version of the Android SDK.
- **OSMDroid Map Library**: an open source mapping library that replaces Android Native MapView. It allows users to implement an abstract tile provider for offline or online data and also to plot overlays over the map, like icons, tracking locations and drawing geometries.
- **Java Open Mobility Library**: an open source library that allows the creation, insertion, query and update of geospatial vector features and raster tiles on OGC GeoPackage Standard for Android applications. It uses JTS Topology Suite and GeoAPI to stores data on OGC Simple Features Specification.

4. ACKNOWLEDGMENTS

Our thanks to Boeing Research and Technology – Brazil (BR&TB) that is funding the development of the TerraMobile App under the INPE-Boeing project "Geotechnologies for Efficient Energy Crop Management".

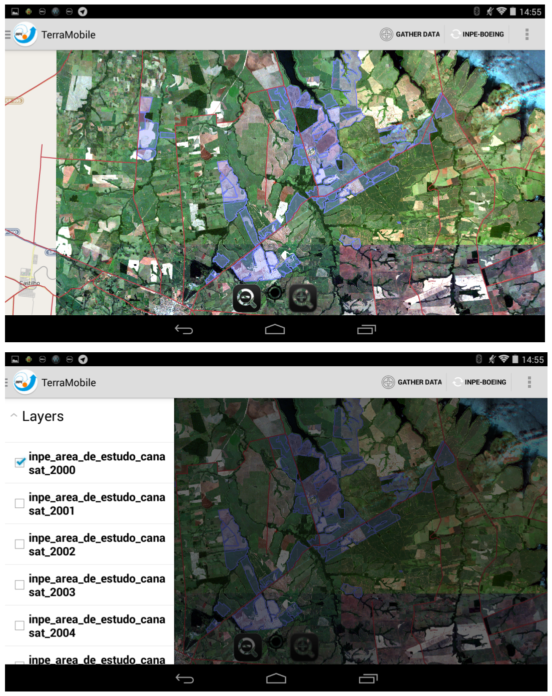

TerraMobile
GATHER DATA
INPE-BOEING
Layers
inpe_area_de_estudo_cana sat_2000
inpe_area_de_estudo_cana sat_2001
inpe_area_de_estudo_cana sat_2002
inpe_area_de_estudo_cana sat_2003
inpe_area_de_estudo_cana sat_2004
14:55

Figure 2 – TerraMobile App.

5. REFERENCES

[1] Tsou, M. H. 2004. *Integrated mobile GIS and wireless internet map servers for environmental monitoring and management*. In: Special issue on Mobile Mapping and Geographic Information Systems, Cartography and Geographic Information Science 31 (3): 153–165.

[2] Poorazizi, E., Alesheikh, A. A. and Behzadi, S. 2008. *Developing a Mobile GIS for Field Geospatial Data Acquisition*. Journal of Applied Sciences, 8(18), 3279-3283.

[3] INPE. 2014. *Monitoramento da Floresta Amazônica Brasileira por Satélite* (*Monitoring the Brazilian Amazon Forest by Satellite*). Available at www.obt.inpe.br/prodes.

[4] OGC. 2006. "OpenGIS Web Map Server Implementation Specification". Available at: http://www.opengeospatial.org/

[5] OGC. 2010. "OpenGIS Web Feature Service 2.0 Interface Standard". Available at: http://www.opengeospatial.org/

[6] Rajabifard, A., Feeny and M. E., Williamson, I. 2002. *Future Directions for SDI Development*. International Journal of Applied Earth Observation and Geoinformation 4 (1), 11-22.

[7] OGC. 2014. *GeoPackage Encoding Standard*. Available at: http://www.opengeospatial.org/

[8] Camara, G.; Vinhas, L.; Queiroz, G. R.; Ferreira, K. R.; Monteiro, A. M. V.; Carvalho, M. T. M. and Casanova, M. A. 2008. *TerraLib: An open-source GIS library for large-scale environmental and sócio-economic applications*. Open Source Approaches to Spatial Data Handling. Berlin: Springer-Verlag

CLAPP: Characterizing Loops in Android Applications (Invited Talk)

Yanick Fratantonio
UC Santa Barbara, USA
yanick@cs.ucsb.edu

Aravind Machiry
UC Santa Barbara, USA
machiry@cs.ucsb.edu

Antonio Bianchi
UC Santa Barbara, USA
antoniob@cs.ucsb.edu

Christopher Kruegel
UC Santa Barbara, USA
chris@cs.ucsb.edu

Giovanni Vigna
UC Santa Barbara, USA
vigna@cs.ucsb.edu

ABSTRACT

When performing program analysis, *loops* are one of the most important aspects that needs to be taken into account. In the past, many approaches have been proposed to analyze loops to perform different tasks, ranging from compiler optimizations to Worst-Case Execution Time (WCET) analysis. While these approaches are powerful, they focus on tackling very specific categories of loops and known loop patterns, such as the ones for which the number of iterations can be statically determined.

In this work, we developed a static analysis framework to characterize and analyze *generic* loops, without relying on techniques based on pattern matching. For this work, we focus on the Android platform, and we implemented a prototype, called CLAPP, that we used to perform the first large-scale empirical study of the usage of loops in Android applications. In particular, we used our tool to analyze a total of 4,110,510 loops found in 11,823 Android applications, and we gained several insights related to the performance issues and security aspects associated with loops.

Categories and Subject Descriptors

D.4.6 [**Software Engineering**]: Security and Protection

Keywords

Android, Static Analysis, Loop Analysis

1. INTRODUCTION

Over the past few decades, there has been an explosion in the development and application of program analysis techniques to achieve a variety of goals. Program analysis has been used for compilation and optimization purposes, for studying a variety of program properties, for detecting bugs, vulnerabilities, malicious functionality, and, ultimately, for understanding program behavior. When performing program analysis, one of the most important aspects that needs to be taken into account are *loops*, which are undoubtedly one of the most useful and essential constructs when writing programs. However, they are also one of the most challenging ones to handle: In fact, even answering the simplest questions (e.g., "Is a given loop going to terminate?") is, in the general case, an undecidable problem.

When applying program analysis, loops also have particular importance for optimization or security purposes: the execution of a performance-intensive operation (e.g., a GUI-related operation) or of a security-relevant operation (e.g., file deletion) might not constitute a problem when executed only occasionally, but it could be deemed as problematic when executed multiple times within a loop. In the past, much research has been focused on the analysis of loops, mainly to perform Worst-Case Execution Time (WCET) analysis [2], which aims to statically determine how many times a loop can be executed in the *worst possible case*, and to perform *loop unrolling* [1], which aims to *unwind* loops' execution to gain a performance boost. While these approaches are powerful, they rely on pattern matching or focus on handling only very specific types of loops.

In this work[1], we developed a novel loop analysis framework (based on static analysis) to characterize loops under many different aspects, such as how they are controlled, which operations they perform, and their impact under both the performance and security aspects. In particular, we focused on the analysis of Android applications, and we developed a tool, called CLAPP, which works directly on Dalvik bytecode, and it therefore does not rely on having access to the application's source code. The key advantage of our approach is that it is completely generic and can be applied to any kind of program. Moreover, our approach does not rely on the identification of *known* cases through techniques based on pattern matching.

We used CLAPP to perform the first large-scale empirical study on 4,110,510 loops contained in 11,823 distinct Android applications. The results allowed us to study the different use cases for writing loops in Android applications, and, more in general, to characterize the usage of loops under two main perspectives, performance and security.

2. LOOP ANALYSIS FRAMEWORK

Our loop analysis framework is constituted by several analysis steps. First, the analyzer unpacks the given Android

[1]The full version of this paper has been published in FSE 2015 [3].

DeMobile'15, August 31, 2015, Bergamo, Italy
ACM. 978-1-4503-3815-8/15/08
http://dx.doi.org/10.1145/2804345.2804355

application, and it parses the Dalvik bytecode into a custom intermediate representation (in SSA form) suitable for performing static analysis. As a second step, the analyzer identifies all loops defined in the application, and it then performs the analyses at the core of our approach, *control analysis* and *body analysis*.

Control Analysis. This analysis aims to determine whether the number of loop's iterations is bounded, and, more in general, to characterize the factors that control it. This is achieved by first identifying all exit paths and exit conditions. Then, for each register involved in each condition, the analyzer constructs an *expression tree* that encodes the operations that *initialize* the register's value before the first iteration of the loop, and that *update* it during each iteration. Then, the analysis performs selective abstract interpretation to determine what is the *trend* of each register's value, attempting to answer questions such as "Is the value constant?" and "Is the value going to eventually increase?". As we discuss in the full version of this paper [3], the answers to these questions proved to be of key importance to answer termination-related questions, and to characterize which kind of external factors can influence the number of iterations of a given loop.

Body Analysis. This analysis aims to characterize the operations executed for each loop's iterations. For each loop, the analysis computes the set of framework APIs that could be potentially invoked within the loop's body. This is done by first computing an over-approximation of the callgraph, and by then performing reachability analysis. This set of methods characterizes the intent of a given loop, and it allows us to perform subsequent powerful analyses, such as the identification of problematic loops.

3. EVALUATION

This section discusses the large-scale empirical study we performed, the results we obtained, and the insights we gained.

Dataset. Our dataset is constituted by 15,240 applications selected, at random, from the ones collected by the PlayDrone project [4].

Overall Results. Among the 15,240 applications selected for the experiments, our prototype was able to successfully analyze 11,823 (77.57%) of them. The analysis of the remaining 3,417 applications did not terminate before the timeout (30 minutes per app). For the applications that were successfully processed, our tool analyzed a cumulative total of 4,110,510 loops, and identified a total of 118,190,014 API framework methods that could potentially be invoked in these loops. On average, analyzing each application takes 96.77 seconds, and analyzing each loop takes 50.86 seconds.

Control Analysis Results. We now report the results related to the control aspect of loops. Our analysis identified 3,196,119 (77.70%) *simple* loops (i.e., loops with only one exit path with one condition) and 910,841 (22.22%) *complex* loops (i.e., loops with one or more exit paths with several exit conditions). For the 3,550 (0.08%) remaining loops, our analysis determined that there were no explicit exit paths, which might indicate the presence of infinite loops. As another interesting statistic, we found that 266,667 (6.48%) of the loops contain at least one nested loop. Our analysis also determined that 2,601,240 (63.28%) of the loops are guaranteed to terminate, and that all the exit paths associated to 6,256 loops do not seem to be satisfiable, thus once again indicating the presence of potentially-infinite loops. Our analysis also classified 24,842 (0.60%) loops as *risky*, with which we refer to loops that, independently from whether they terminate or not, a subtle change in their body might cause them to become infinite. As a clarifying example, consider the loop "`for (i=0; i != 12; i+=3){...}`": this loop will iterate exactly four times. However, a modification to how the variable `i` is updated could suddenly introduce an infinite loop.

Body Analysis Results. The results of the body analysis show that, in most cases, developers make use of loops to invoke low-risk APIs. For example, they use loops to perform simple iterations over app-specific objects, perform cryptographic operations, generating random numbers, parsing data, and iterating over different data structures. However, our analysis also identified 1,057,628 loops that could potentially invoke network-related API functions, 764,240 of which could be executed by the app's main UI thread. The Android official documentation clearly states that no potentially-blocking operations should be ever performed within the main UI thread, since, in certain scenarios, the app might be terminated with the infamous Application Not Responding (ANR) error message. This aspect is so problematic that a recent version of Android introduced `StrictMode`, which is, quoting the official documentation, a tool to "catch accidental disk or network access on the application's main thread." However, this mechanism can be explicitly disabled by an Android application, and, interestingly, we found 207,888 loops that potentially do so. We invite the interested reader to consult the detailed results reported in the full version of this paper [3].

4. CONCLUSIONS

We presented CLAPP, a tool that implements a general loop analysis framework (based on static analysis) to study a variety of aspects related to the usage of loops in Android applications. We used CLAPP to perform the first large-scale empirical study on 4,110,510 loops contained in 11,823 Android apps, and we gained several insights related to their control, body, performance, and security aspects.

5. REFERENCES

[1] D. Berlin, D. Edelsohn, and S. Pop. High-level Loop Optimizations for GCC. In *Proceedings of the GCC Developers Summit*, 2004.

[2] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis. In *Workshop on Worst-Case Execution Time Analysis (WCET)*, 2007.

[3] Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna. CLAPP: Characterizing Loops in Android Applications. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE)*, 2015.

[4] N. Viennot, E. Garcia, and J. Nieh. A Measurement Study of Google Play. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRIC)*, 2014.

Optimizing Display Energy Consumption for Hybrid Android Apps (Invited Talk)

Ding Li, Angelica Huyen Tran, and William G. J. Halfond
University of Southern California
Los Angeles, California, USA
{dingli, tranac, halfond}@usc.edu

ABSTRACT

Energy has emerged as an important quality metric for apps that run on mobile platforms. This talk describes our approach for reducing display energy by automatically changing the color schemes used by a web app so that the pages consume less energy when displayed on an OLED based smartphone.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Diagnostics

General Terms

Performance

Keywords

Energy optimization, display energy, mobile systems, web applications

1. INTRODUCTION AND MOTIVATION

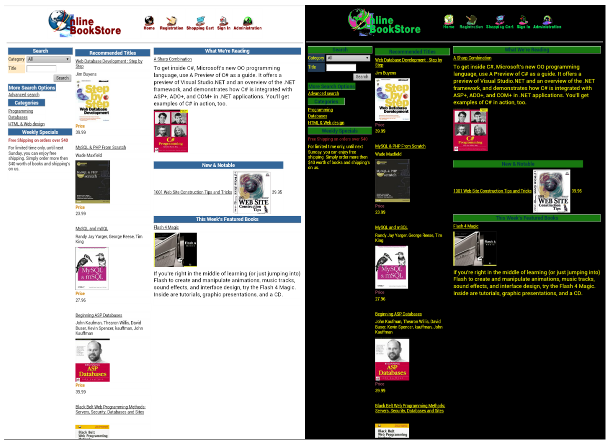
Energy has become an important quality metric for mobile apps. Recent studies have shown that energy related complaints can represent a significant source of user unhappiness with an app [2]. However, developers have traditionally not had extensive guidance for how to reduce the energy consumption of their apps. New techniques (e.g., [3, 4, 8]) have started to address this problem by providing developers with tools to understand, at a source line level, what is consuming energy in their apps.

These techniques have enabled us to gain new insights into how energy is consumed by apps and identify areas for improvement [7]. In particular, these results show that almost 62% of an app's energy is consumed when it is either sleeping or waiting for input. The primary cause of that non-execution time energy consumption is display energy, which can consume 40–60% of an app's total energy [9]. These results motivate attention to reducing display energy.

DeMobile'15, August 31, 2015, Bergamo, Italy
ACM. 978-1-4503-3815-8/15/08
http://dx.doi.org/10.1145/2804345.2804356

Many popular modern smartphones use OLED based displays. An interesting characteristic of these displays is that they consume less energy when displaying dark (e.g., black) colors than light colors (e.g., white). Intuitively, a way to take advantage of this is to modify an app's colors to use a color layout with black as the dominant color instead of the ever popular light or white backgrounds. However, this is challenging because the UIs of an app can be dynamically generated and colors must be adjusted in a way that balances energy efficiency and aesthetics. Prior work has either required developers to manually generate these new color schemes, which is labor intensive, or simply used color inversion, which reduces aesthetics since color differences are not maintained in the transformed app [1].

In this talk, we present our prior work in developing an approach that can help developers in making their web apps more energy efficient when displayed on an OLED based smartphone [9]. The way this is done is that the approach analyzes the implementation of the target web application to identify its UI layout and colors. Then the approach uses this information to find a new energy efficient color scheme that tries, as much as possible, to maintain color differences between neighboring elements. An example before and after image of a transformed web app is shown in Figure 1.



Free Shipping on orders over $40
For limited time only, until next Sunday, you can enjoy free shipping. Simply order more then $40 worth of books and shipping's on us.

Web Database Development : Step by Step
Jim Buyens
Price
39.99

MySQL & PHP From Scratch
Wade Maxfield
Price
23.99

MySQL and mSQL
Randy Jay Yarger, George Reese, Tim King
Price
27.96

Beginning ASP Databases
John Kaufman, Thearon Willis, David Buser, Kevin Spencer, kauffman, John Kauffman
Price
39.99

Black Belt Web Programming Methods: Servers, Security, Databases and Sites

A Sharp Combination
To get inside C#, Microsoft's new OO programming language, use A Preview of C# as a guide. It offers a preview of Visual Studio.NET and an overview of the .NET framework, and demonstrates how C# is integrated with ASP+, ADO+, and COM+ in .NET applications. You'll get examples of C# in action, too.

1001 Web Site Construction Tips and Tricks 39.95

Flash 4 Magic
If you're right in the middle of learning (or just jumping into) Flash to create and manipulate animations, music tracks, sound effects, and interface design, try the Flash 4 Magic. Inside are tutorials, graphic presentations, and a CD.

Figure 1: Example output of the Nyx approach

2. APPROACH

Our approach can be roughly described as having five steps. The first step is a static analysis that examines the server-side code of the web app and builds a model representing the potential HTML content that it could generate. In the second step, the approach parses this model to identify the visual relationships between tags, such as "contained by" and "next to." After identifying these relationships, the third step of the approach is to determine the colors of the text and background of the different HTML elements and abstract this information into a Color Conflict Graph (CCG) that only shows colors and their visual relationships with each other. Each edge of the CCG is weighted to signify a priority in maintaining the corresponding relationship. Once the CCG is complete, the approach solves for a recoloring that is more energy efficient but maintains, as closely as possible, the color differences between edges in the CCG. Since solving for the best recoloring is an NP problem, in the fourth step, the approach uses a search-based approximation to find an optimal solution. Finally, in the fifth step, the approach uses bytecode rewriting and parsing-based transformations to insert the new color scheme into the application.

3. EVALUATION

To evaluate the effectiveness of our approach, we implemented it in a prototype tool, Nyx [10]. Our implementation is for Java-based web applications and we evaluated it on a set of seven web applications that ranged in size from 5K to 154K SLOC and used a variety of frameworks, such as JSP, Velocity, and Turbine. For all of the apps it was possible to analyze and transform them in under three minutes.

In terms of energy savings, we found that energy consumption of the display, once it had loaded and rendered the page, decreased by 40%. We also saw a power decrease of 25% while the page was loading and rendering. This occurred because modern mobile browsers start to render a page before it has finished loading. Overall, these are significant results and show the potential of our technique to reduce energy consumption.

We also performed a study to assess the aesthetics of the transformed web pages. To do this, we gave a group of 20 grad students the original and optimized versions of the subject apps and asked them to rate their attractiveness, readability, and acceptability. Overall, the students rated the transformed web pages as 17% less attractive and 14% less readable (using a 10 point scale). However, when given a summary of the energy savings that could be achieved for each page, 67% of the students indicated it would be acceptable for normal usage, and 97% indicated it would be acceptable when the battery level was critical.

4. SUMMARY

The energy consumed by the display component of a smartphone device represents a significant portion of the overall energy expended by the device. This talk focused on recent work ([9]), Nyx, that optimizes energy consumption by automatically transforming the colors used by an app's UI so that the UI consumes less energy when displayed on an OLED based smartphone. The evaluation results of Nyx were very positive and show that it was able to reduce display energy by an average of 40%. Nyx was able to generate these results in an average time of under three minutes and the aesthetics of its color schemes were acceptable to users when they were made aware of the power savings.

In future work, we are investigating the use of Nyx to detect display energy hotspots (e.g., [?]) and refine the Nyx mechanisms to improve on the aesthetics for parts of the transformation that were considered less attractive by end users. We will also investigate other areas for energy improvement in mobile apps, such as network communication [5, 6].

5. ACKNOWLEDGMENTS

The work described in this talk was supported by National Science Foundation grant CCF-1321141.

6. REFERENCES

[1] M. Dong and L. Zhong. Chameleon: A Color-adaptive Web Browser for Mobile OLED Displays. In *MobiSys*, 2011.

[2] J. Gui, S. Mcilroy, M. Nagappan, and W. 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 (ICSE)*, May 2015.

[3] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating android applications' cpu energy usage via bytecode profiling. In *Proceedings of the First International Workshop on Green and Sustainable Software (GREENS)*, pages 1–7, May 2012.

[4] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, May 2013.

[5] D. Li and W. G. Halfond. Optimizing energy of http requests in android applications. In *Proceedings of the Third International Workshop on Software Development Lifecycle for Mobile (DeMobile) – Short Paper*, September 2015. To Appear.

[6] D. Li and W. G. J. Halfond. An investigation into energy-saving programming practices for android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software (GREENS)*, June 2014.

[7] D. Li, S. Hao, J. Gui, and W. G. Halfond. An empirical study of the energy consumption of android applications. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, September 2014.

[8] D. Li, S. Hao, W. G. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2013.

[9] D. Li, A. H. Tran, and W. G. J. Halfond. Making Web Applications More Energy Efficient for OLED Smartphones. In *Proceedings of the International Conference on Software Engineering (ICSE)*, June 2014.

[10] D. Li, A. H. Tran, and W. G. J. Halfond. Nyx: A display energy optimizer for mobile web apps. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE) – Tool Track*, September 2015. To Appear.

Author Index