UNIVERSITY OF CALIFORNIA
RIVERSIDE

Application of Software Analysis in Detecting Vulnerabilities: Testing and Security Assessment

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Arash Alavi

September 2019

Dissertation Committee:

Dr. Rajiv Gupta, Co-Chairperson
Dr. Zhiyun Qian, Co-Chairperson
Dr. Srikanth V. Krishnamurthy
Dr. Zhijia Zhao

The Dissertation of Arash Alavi is approved:

Committee Co-Chairperson

Committee Co-Chairperson

University of California, Riverside

Acknowledgments

The work presented in this thesis would not have been possible without the inspiration, support, and help of a number of wonderful individuals. Of course, I would like to start by sending my greatest respect and thankfulness to my Ph.D. adviser Prof. Rajiv Gupta. Without his guidance and inspiration, this journey would not have been possible. He provided me with a tremendous degree of freedom and many opportunities over the years. Throughout the course of my Ph.D. research, I have learned extensively from him, from his constructive direction and wisdom to his passion and enthusiasm, *Thank you Prof. Gupta.*

I would also like to express my gratitude and appreciation to my co-supervisor, Prof. Zhiyun Qian for all of his support and help. His critical suggestions and advice at every stage of my graduate research have been very valued.

I would like to express my most sincere appreciation and special thanks to Prof. Iulian Neamtiu for his fundamental role in my doctoral work and helping me on this journey from the very beginning. I feel extremely lucky to have had the chance to work together on several projects.

I'm very thankful to my committee members Prof. Srikanth V. Krishnamurthy and Prof. Zhijia Zhao for their valuable feedback and support in various parts of my work.

I am grateful to have worked with intellectual and awesome lab mates and colleagues. Yongjian Hu and Tanzirul Azim have simultaneously been a friend, mentor, and co-author, and I enjoyed a lot working with them. I would like to thank Alan Quach, Abbas Mazloumi, Chengshuo (Bruce) Xu, Xiaolin Jiang, Pritom Ahmed, Zachary Benavides, Keval Vora, Farzad Khorasani, Vineet Singh, Amlan Kusum, and Bo Zhou.

I am also so grateful of my colleagues at Stanford University, Dr. Amir Bahmani, Prof. Michael P. Snyder, Dr. Xiao Li, and Keith Bettinger. The internship with them was a very nice experience and helped me to improve in research and coding.

I would like to thank the NSF for their grants to Prof. Gupta, Prof. Neamtiu, and Prof. Qian which partially supported my research.

Finally, my love, Ghazal and my family receive my deepest gratitude and love for their dedications and many years of unconditional and endless supports. Words are not enough to express my gratitude to my love, parents, sister, grand father and grand mother.

To my love Ghazal and my parents for their endless love

ABSTRACT OF THE DISSERTATION

Application of Software Analysis in Detecting Vulnerabilities:
Testing and Security Assessment

by

Arash Alavi

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2019
Dr. Rajiv Gupta, Co-Chairperson
Dr. Zhiyun Qian, Co-Chairperson

With the increasing complexity of application software there is an acute need for software analysis approaches that are capable of identifying bugs, failures, and most importantly vulnerabilities on a large scale. In this dissertation, first we stress the necessity of having automated software analysis approaches and then propose analysis approaches for detecting vulnerabilities in software via analysis and testing in general, and security assessment in particular. We show the efficiency and effectiveness of these analysis techniques in detecting vulnerabilities.

First, we study security issues in smartphone applications by studying the security discrepancies between Android apps and their website counterparts, depicting the essential need of efficient software analysis techniques to fully automate the mobile app analysis process. By a comprehensive study on 100 popular app-web pairs, we find that, with respect to various security policies, the mobile apps often have weaker or non-existent security measures compared to their website counterparts.

Second, as a consequence of the former, we develop AndroidSlicer, the first novel, efficient, and effective dynamic program slicing tool for Android apps that is useful for a variety of tasks, from testing to debugging to security assessment. Our work in this domain focuses on making large scale applications of slicing practical in order to detect bugs and vulnerabilities in real-world apps. We present two new applications of the dynamic slicing technique in mobile apps: (1) detecting the "stuck" states (missing progress indicators) in mobile apps. We present, implement, and evaluate ProgressDroid, a tool for discovering missing progress indicator bugs based on program dependencies; and (2) detecting security vulnerabilities in unique device ID generators.

Finally, in the same vein of deploying analysis tools for detecting vulnerabilities, we present *GAGA*, an efficient genetic algorithm for graph anonymization that simultaneously delivers high anonymization and utility preservation. Experiments show that *GAGA* improves the defense against DA techniques by reducing the rate of successfully de-anonymized users by at least a factor of 2.7× in comparison to the baseline and at the same time, under 16 graph and application utility metrics, *GAGA* is overall the best at preserving utilities.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many of the existing research works for detecting vulnerabilities still rely on significant manual efforts (i.e., reverse engineering, trial and error, and hacky workarounds) [66, 117, 36, 17]. On the other hand, while there are various security analysis tools which engineers, developers, and researchers have at their disposal, some of them are insufficient for effective automated vulnerability detection [122, 29, 55, 119, 121]. Hence, improving the array of software analysis tools in order to deploy effective and efficient software analysis approaches capable of making the vulnerability detection process far easier, widespread, and automated is essential.

This research addresses the challenges in detecting vulnerabilities via automated software analysis approaches. An overview is shown in Figure 1.1. Detected vulnerabilities can be a bug which can lead to a security breach, a wrong implementation of a security policy, or a weak defense mechanism. We stress the necessity of having automated software analysis approaches and then propose analysis approaches for detecting vulnerabilities.

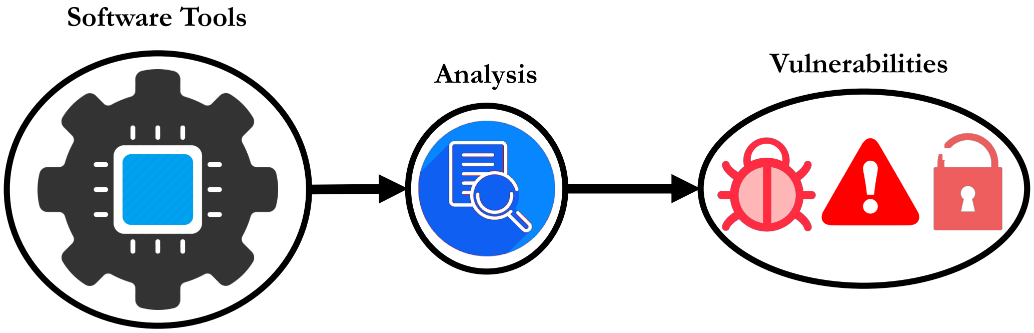


Figure 1.1: Detecting vulnerabilities via software analysis

1.1 Security Vulnerabilities in Android Apps

Nowadays, users appear to be substituting websites for mobile applications which may be more convenient to access throughout the day. Given that a large number of services already exist and are offered as traditional websites, it is expected that many apps are basically remakes or enhanced versions of their website counterparts. Examples of these include mobile financial applications for major banking corporations like Chase and Wells Fargo or shopping applications like Amazon and Target. The software stack for the traditional web services has been well developed and tested for many years. The security features are also standardized (*e.g.,* cookie management and SSL/TLS certificate validation). However, as the web services are re-implemented as mobile apps, many of the security features need to be re-implemented as well. This can often lead to discrepancies between security policies of the websites and mobile apps. As demonstrated in a recent study [37], when the standard feature of SSL/TLS certificate validation logic in browsers is re-implemented on mobile apps, serious flaws are present that can be exploited to launch MITM (Man-In-The-Middle) attacks. Such an alarming phenomenon calls for a more comprehensive analysis of aspects beyond the previous point studies.

Hence, We aim to identify security vulnerabilities in mobile Apps by presenting security discrepancies in mobile apps and their website counterparts [17]. Specifically, we examine a number of critical website security policies that need to be re-implemented in mobile apps. For this purpose, we explore the top 100 popular Android apps from various categories in Google Play, as well as their website counterparts to perform a comprehensive study about their security discrepancies on their security policies. As expected, we observe that often the app security policies are much weaker than their website counterparts.

1.2 Dynamic Slicing for Android

Our semi-automated approaches in the above work stresses the necessity of having more automated and efficient software analysis tools to help detect vulnerabilities. Therefore, as our next contribution is a dynamic slicing approach for mobile applications [24] – the choice of dynamic slicing is motivated by its versatility in automating variety of analyses ranging from testing to debugging to security assessment. The key contribution of this work is the design and implementation of the first novel, efficient, and effective dynamic program slicing tool for Android apps. While dynamic slicing has targeted traditional applications running on desktop/server platforms, our work brings dynamic slicing to Android. This is challenging for several reasons. First main challenge is due to asynchronous callback constructions and the IPC-heavy environment. Second, the sensor-driven, timing-sensitive nature of the Android platform poses a significant challenge as it requires that dynamic slicing entail minimal overhead. Any instrumentation system to support capturing relevant system and app state should be lightweight. To address the above challeges, we introduce

AndroidSlicer[1], the first slicing approach for Android. We present three conventional applications of AndroidSlicer that are relevant in the mobile domain: (1) finding and tracking input parts responsible for an error/crash; (2) fault localization, i.e., finding the instructions responsible for an error/crash; and (3) reducing the regression test suite. Experiments with these applications show that AndroidSlicer is *effective* and *efficient*.

1.3 New Applications of Dynamic Slicing in Android

We present two new applications of dynamic slicing in mobile apps domain. The first facilitates testing Android apps from the perspective of one of the important user interface design principles – always showing a progress indicator to the user for *long-running* operations. The second helps identify a specific type of vulnerability in approaches for forming unique device identification signature.

User interface guidelines often emphasize the importance of using progress indicators [5, 6, 91] for long-running operations such as network communications. To the best of our knowledge, there is no study addressing the “missing progress indicators” for heavy-weighted network operations in mobile apps. Hence in the first application, we try to automatically find “missing progress indicators” based on program semantics, in particular program dependencies using our dynamic slicing technique.

In the second application, we identify the potential vulnerabilities in unique device identification approaches in mobile apps. There are many use cases where the mobile app developers need an unique ID to identify Android devices. In this work, we focus on

[1] https://github.com/archer29m/AndroidSlicer

tracking the apps installations approaches where the developers try to identify unique fresh installations on users devices. One use case of such an approach is where the developers want to know the number of devices that have installed their apps through a specific app promoter channel. Another use case is related to apps with initial discounts. We show the vulnerabilities in both use cases which lead to financial loss.

1.4 Efficient Genetic Algorithm for Graph Anonymization

In the next section of this thesis, in the same direction of deploying analysis tools for detecting vulnerabilities, we address the problem of user data privacy preservation in graph data (e.g., social networks). The main drawback of existing anonymization techniques is that they trade-off anonymization with utility preservation. To address this limitation, we propose, implement, and evaluate *GAGA*, an efficient genetic algorithm for graph anonymization [16]. Our results show that *GAGA* is highly effective and has a better trade-off between anonymization and utility preservation compared to existing techniques.

1.5 Thesis Organization

The dissertation is organized as follows. In Chapter 2, we study security discrepancies between mobile apps and their website counterparts. Chapter 3 discusses our dynamic slicing technique. Next in Chapter 4, we present two new applications of dynamic slicing that identify a new kind of bug and a new vulnerability in mobile apps. In Chapter 5, we present our new efficient graph anonymization technique to preserve user privacy in graph data. Finally in Chapter 6, we provide the conclusion and discuss the possible future work.

Chapter 2

Security Vulnerabilities in Android Apps

Many web services are now delivered via mobile apps. It is expected that many apps are basically remakes or enhanced versions of their website counterparts. The re-implementation of many of these web services in mobile apps, can often lead to discrepancies between security policies of the websites and mobile apps. Traditional web services have been well developed, tested, and standardized for many years but the mobile apps are newly developed and may naturally lack the maturity of web services. Hence, our hypothesis is that many security policies in mobile apps are significantly weaker than those in traditional website environment. To verify our hypothesis, we study the top 100 popular Android apps (each of which has more than 5,000,000 installs at the time of the study) from various categories in Google play, as well as their website counterparts, to perform a comprehensive study about their security discrepancies.

We study the security discrepancies between apps and their websites from the following 3 main domains: authentication policies, cookie management security policies, and use of libraries. We identify a set of critical security policies that are commonly employed by app/web service pairs. Since such pairs represent essentially the same services, the discrepancy in security policies effectively lowers the security of the overall service.

2.1 Background

In this section, we provide some background knowledge of three main security policies that we use to compare the implementations of those policies in mobile apps and their websites. We begin with the introduction to different authentication related security policies, and then we discuss the storage encryption methods that are used in mobile apps and by different browsers. Finally, we give a brief overview of library use in Android apps and how it differs from the browser scene.

2.1.1 Authentication Security Policies

We anticipate to see many different forms of authentication security policies in place for both apps and websites. One of the most common forms of authentication policies that can be seen are CAPTCHAs. Others include a mandatory wait period or denial of access either to an account or service. All three of these have potential to be IP/machine-based or globally user-based.

CAPTCHA. Though CAPTCHAs are designed with the purpose of defeating machines, prior research has shown that they can be defeated by machines algorithmically

[77] or via speech classification [100]. Due to the possibility of CAPTCHA replay attacks, Open Web Application Security Project (OWASP) recommends that CAPTCHA be only used in “rate limiting” applications due to text-based CAPTCHAs being crackable within 1-15 seconds [83].

Waiting Time. A less common method of authentication policy is the usage of waiting periods to limit the number of logins that can be attempted. The response is in the form of an explicit message or disguised through a generic “Error” message. Waiting periods, either for a single IP or for the user account is a very effective method to slow down and mitigate aggressive online credential guessing attacks. Depending on the implementation, it may operate on a group of IPs (*e.g.,* belonging to the same domain).

Denial of Access. An extreme policy is the denial of access, where an account is essentially “locked” and additional steps are necessary to regain access (*e.g.,* making a phone call) [111]). If an attacker knows the login ID of an account, then he can lock the account by repeatedly failing the authentication. Though denial of access is very secure against online password guessing attacks, OWASP recommends that such method be used in high-profile applications where denial of access is preferable to account compromises [82].

2.1.2 Storage Encryption Methods

Browsers on PCs by default encrypt critical data for long term storage. In the case of Chrome on Windows, after a successful login into a website, by clicking “Save Password”, the browser stores the password in encrypted form using the Windows login credential as the key. It is not the same for mobile apps. For instance, the APIs for managing cookies do not require the cookies to be encrypted.

2.1.3 Libraries

Mobile apps use libraries for different functionalities such as advertisements, audio and video streaming, or social media. Previous studies [43, 30, 8] have shown security and privacy issues that arise by use of some libraries which can lead to leakage of sensitive user information, denial-of-service, or even arbitrary code execution. For services delivered through websites on the other hand, no website-specific native libraries are loaded. Unlike libraries embedded in apps that may be out-of-date and vulnerable, libraries used in browsers (*e.g.,* flash) are always kept up-to-date and free of known vulnerabilities.

2.2 Related Work

As far we know, there are no in depth studies that explicitly analyze the similarities and differences between mobile applications and their website counterparts in terms of security. Fahl et al. [37] understood the potential security threats posed by benign Android apps that use the SSL/TLS protocols to protect data they transmit. Leung et al. [62] recently studied 50 popular apps manually to compare the Personally Identifiable Information (PII) exposed by mobile apps and mobile web browsers. They conclude that apps tend to leak more PII (but not always) compared to their website counterparts, as apps can request access to more types of PII stored on the device. This is a demonstration of the discrepancy of privacy policies between apps and websites. In contrast, our work focuses on the discrepancy of security (not so much privacy) policies between apps and websites. Zuo et al. [123] automatically forged cryptographically consistent messages from the client side to test whether the server side of an app lacks sufficient security layers. They applied their techniques to test the server side implementation of 76 popular mobile apps with 20 login attempts each and conclude that many of them are vulnerable to password brute-forcing attacks, leaked password probing attacks, and Facebook access token hijacking attacks. Sivakorn et al. [93] recently conducted an in-depth study on the privacy threats that users face when attackers have hijacked a user's HTTP cookie. They evaluated the extent of cookie hijacking for browser security mechanisms, extensions, mobile apps, and search bars. They observed that both Android and iOS platforms have official apps that use unencrypted connections. For example, they find that 3 out of 4 iOS Yahoo apps leak users' cookies.

2.3 Methodology and Implementation

In this section we describe our methodology and implementation details of our approach to analyze app-web pairs. We selected the top 100 popular Android apps (each of which has more than 5,000,000 installs) from popular categories such as shopping, social, news, travel & local, etc. in Google play. All apps have a corresponding website interface that offers a similar functionality. For each app-web pair, we created legitimate accounts using default settings. This was done to mimic the processes of an actual user interacting with an app or website.

2.3.1 Login Automation Analysis

We automate logins and logging for apps and websites for the purposes of this study. For each app-web pair, we perform 101 login attempts automatically using randomly generated alphanumeric passwords for the first 100 attempts followed by an attempt with the correct password. 100 attempts was chosen as this was an order of magnitude larger than what an average user would perform within a span of 24 hours [18] so that we can identify the polices when the number of lognin attempts goes much beyond the average frequency of attempts in real world. Allowing unlimited number of login attempts is a security vulnerability because it allows an attacker to perform brute force or dictionary attacks. Another security aspect of login attempts is that if the system leaks the user ID (*e.g.,* email) during the login authentication checking, by returning error messages such as "wrong password" either in the UI or in the response message, then an attacker can send a login request and learn whether a user ID has been registered with the service. Therefore, we also compare the servers' responses to login requests, either shown in the UI or found in the response packet, for both apps and websites.

2.3.2 Sign up Automation Analysis

Besides login tests, we perform the sign up tests that can also potentially leak if the username has been registered with the service. Again, we simply need to compare the servers' responses to sign up requests for apps and websites. For both login and sign up security policies, if a service where the website allows for only a limited number of logins/sign-ups before a CAPTCHA is shown whereas the mobile app never prompts with a CAPTCHA, an attacker would be inclined to launch an attack following the mobile app's protocol rather than the website's. Test suites for the purposes of testing mobile apps and websites were created using *monkeyrunner* and *Selenium Webdriver*, respectively.

2.3.3 Authentication Throughput Analysis

From the login automation analysis, we collect the set of app-web pairs where we find different behaviors between the app and the website counterpart, we call this set "discrepancy list". Using the network traffic monitoring tools *Fiddler* and *mitmproxy*, we log network traffic traces for all app-web pairs in the discrepancy list. Using the information in the network traffic traces, we analyze how authentication packets are structured for each client as well as finding what sort of information is being shared between a client and server. This enables us to determine whether the app-web pair has the same authentication protocol and share the same set of backend authentication servers. In addition, this allows us to construct tools capable of sending login request packets without actually running the mobile app, pushing for higher throughput of authentication attempts. The tool also logs all responses received from a server. To push the throughput even further, we can optionally parallelize the login requests (from the same client) by targeting additional backend authentication server IPs simultaneously. Our hypothesis is that the throughput can be potentially multiplied if we target multiple servers simultaneously.

2.3.4 IP-Changing Clients Analysis

Using *VPN Gate* and a sequence of 12 IP addresses from different geographical locations, including 3 from North America and 9 from other countries, we test the apps and websites regarding their response to accounts being logged in from multiple locations separated by hundreds of miles in a short span of time. The motivation of this analysis was to determine whether app/website has a security policy against IP changes can indicate session hijacks [35]. If not, then an attacker can use the hijacked cookies anywhere without being recognized by the web service. For example an attacker can use a stolen cookie from an app with any IP address to obtain personal and/or financial information pertaining to the user account.

2.3.5 Cookie Analysis

For each app-web pair, we analyze the cookies that are saved on the phone/PC. We collect all the cookies and analyze cookie storage security policies to find whether they are stored in plaintext and more easily accessible. We also perform expiration date comparison testing on 18 shopping app-web pairs from our list of app-web pairs. The hypothesis is that mobile apps run on small screens and it is troublesome to repeatedly login through the small software keyboard; therefore the corresponding app's servers will likely have a more lenient policy allowing the cookies to stay functional for longer time periods.

2.3.6 Vulnerable Library Analysis

While both apps and websites execute client-side code, app code has access to many more resources and sensitive functionalities compared to their website counterpart, *e.g.,* apps can read SMS on the device while javascript code executed through the browser cannot. Therefore, we consider the app code more dangerous. Specifically, vulnerable app libraries running on the client-side can cause serious attacks ranging from denial of service (app crash) to arbitrary code execution. Because of this, for each app, we identify if it uses any vulnerable libraries. We conduct the analysis beyond the original 100 apps to 6400 apps in popular categories. Ideally the libraries should be tagged with versions;

unfortunately, we discover that most libraries embedded in Android apps do not contain the version information as part of their metadata. Therefore, in the absence of direct version information, we perform the following steps instead. First, we search the extracted libraries through the CVE database. If there is any library that is reported to have vulnerabilities, we perform two tests to conservatively flag them as vulnerable. First is a simple time test: we check if the last update time of the app is before the release time of patched library. Obviously, if the app is not updated after the patched library is released, then the app must contain a vulnerable library. If the time test cannot assert that the library is vulnerable, we perform an additional test on the symbols declared in the library files. Specifically, if there is a change (either adding or removing a function) in the patched library, and the change is lacking in the library file in question, then we consider it vulnerable. Otherwise, to be conservative, we do not consider the library as vulnerable.

2.4 Observations

We present our results obtained from following the methodology outlined earlier with respect to several security policies.

Security policies against filed login and sign up attempts. By performing login attempts automatically for each pair of app and website, many interesting discrepancies in security policies have been found. Figure 2.1 summarizes the main results for all 100 pairs, considering their latest versions at the time of experiment (summer 2016).

In general, we see that the security policy is weaker on the app side. There are more apps without security policies than websites. We also see that there are significantly

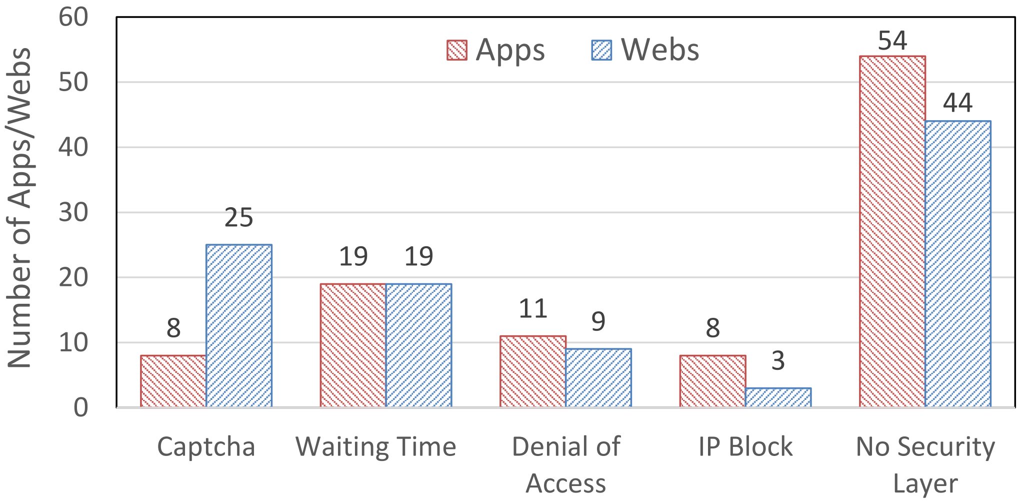


Figure 2.1: Security policies against failed login attempts in apps vs. websites

fewer apps asking for CAPTCHA, presumably due to the concern about usability of the small keyboards that users have to interact with them.

Interestingly, in the case when CAPTCHAs are used both by app and website, the CAPTCHA shown to app users is usually simpler in terms of the number of characters and symbols. For instance, LinkedIn website asks the user to enter a CAPTCHA with 2 words while its app CAPTCHA only has 3 characters. Unfortunately, an attacker knowing the difference can always impersonate the mobile client and attack the weaker security policy. We also observe that more apps employ IP block policies for a short period of time. This is effective against naive online credential guessing attacks that are not operated by real players in the underground market. In reality, attackers are likely operating on a large botnet attempting to perform such attacks, rendering the defense much less effective than it seems. In fact, if the attackers are aware of the discrepancy, they could very well be impersonating the mobile client to bypass stronger protections such as CAPTCHA (which

App-Web	**App Security Layer (App Verrsion)**	**Website Security Layer**	**App Host**	**Website Host**
Babbel	None(5.4.072011) Account lock(5.6.060612)	Account lock	www.babbel.com/api2/login	accounts.babbel.com/en/accounts/sign_in
Ebay	None(3.0.0.19) IP block(5.3.0.11)	Captcha	mobiuas.ebay.com/servicesmobile/v1/UserAuthenticationService	signin.ebay.com/ws/eBayISAPI.dll
Expedia	None(5.0.2)	Captcha	www.expedia.com/api/user/signin	www.expedia.com/user/login
Hotels.com	None(12.1.1.1) IP block(20.1.1.2)	Captcha	ssl.hotels.com/device/signin.html	ssl.hotels.com/profile/signin.html
LivingSocial	None(3.0.2) IP block(4.4.2)	Wait time	accounts.livingsocial.com/v1/oauth/authenticate	accounts.livingsocial.com/accounts/authenticate
OverDrive	None(3.5.6)	Captcha	overdrive.com/account/sign-in	www.overdrive.com/account/sign-in
Plex	None(4.6.3.383) IP block(4.31.2.310)	IP block	plex.tv/users/sign_in.xml	plex.tv/users/sign_in
Quizlet	None(2.3.3)	Wait time	api.quizlet.com/3.0/directlogin	quizlet.com/login
Skype	None(7.16.0.507)	Wait time & Captcha	uic.login.skype.com/login/skypetoken	login.skype.com/login
SoundCloud	None(15.0.15) IP block(2016.08.31-release)	Captcha	api.soundcloud.com/oauth2/token	sign-in.soundcloud.com/sign-in/password
TripAdvisor	None(11.4) IP block(17.2.2)	Captcha	api.tripadvisor.com/api/internal/1.5/auth/login	www.tripadvisor.com/Registration

Twitch	None(4.3.2)
	Captcha(4.11.1)
		Captcha	api.twitch.tv/kraken/oauth2/login	passport.twitch.tv/authorize

We Heart It	None(6.0.0)	Captcha	api.weheartit.com/oauth/token	weheartit.com/login/authenticate
Zappos	None(5.1.2)	Captcha	api.zappos.com/oauth/access_token	secure-www.zappos.com/authenticate

Table 2.1: Discrepancy of authentication policies among app-web pairs.

sometimes requires humans to solve and is hence an additional cost to operate cyber crime).

Table 2.1 lists app-web pairs in detail where apps operate without any security protections whatsoever, at least for the version when we began our study but their websites have some security policies. This allows attackers to follow the app protocol and gain unlimited number of continuous login attempts (confirmed with 1000+ trials). In total, we find 14 such app-web pairs; 8 apps have subsequently strengthened the policy after we notified them. There are however still 6 that are vulnerable to date (that is the time of the study: summer 2016). We also provide a detailed list of all 100 app-web pairs on our project website [9]. To ensure that there is indeed no security protection for these apps, we perform some follow-up tests against the 14 applications and confirm that we could indeed reach up to thousands of attempts (without hitting any limit). Note that our approach ensures that no hidden security policy goes unnoticed (such as the account being silently blocked), as our test always concludes with a successful login attempt using the correct password, indicating that it has not been blocked due to the failed attempts earlier. In the table, we also list the URLs that correspond to the login requests. Since both the domain names and resolved IP addresses (which we did not list) are different, it is a good indication that apps and websites go through different backend services to perform authentications, and hence there are different security policies.

Impact of online credential guessing attacks. To perform online password guessing attacks, one can either perform a brute force or dictionary attack against those possibilities that are deemed most likely to succeed. As an example, the recent leakage of passwords from Yahoo [10] consisting of 200 million entries (without removing duplicates). According to our throughput result, at 600 login attempts per second (which we were able to achieve against some services), one can try every password in less than 4 days against a targeted account (if we eliminate duplicate passwords the number will be much smaller). Let us consider an attacker who chooses the most popular and unique 1 million passwords; it will take less than half an hour to try all of them. Note that this is measured from a single malicious client, greatly lowering the requirement of online password guessing attacks, which usually are carried out using botnets. Another type of attack which can be launched is Denial of Service (DoS) attack. By locking large amount of accounts through repeated logins, attackers could deny a user's access to a service. As we mentioned earlier, we find more apps than websites which have the account lock security policy against the failed authentication (11 apps vs. 9 websites). Account lock security policy is a double edge sword: while it provides security against unauthorized login attempts, it also allows an attacker to maliciously lock legitimate accounts with relative ease. The result shows that this kind of attack can be more easily launched on the app side. We verify this claim against our own account and confirm that we are unable to login with the correct password even from a different IP address.

To perform online username guessing attacks, we report the result of the sign up (registration) security policy testing, which aligns with the login results. We find 5 app-web pairs — 8tracks, Lovoo, Newegg, Overdrive, StumbleUpon — where the app has no security protection against flooded sign up requests while the website has some security protection such as CAPTCHA. We also find that 14 websites leak the user email address during the authentication checking by returning error messages such as "wrong password". In contrast, 17 apps leak such information. The three apps with weaker security policies are AMC Theaters, Babbel, and We Heart It. The discrepancy allows one to learn whether a user ID (e.g., email) has been registered with the service by performing unlimited registration requests. Combined with the password guessing, an attacker can then also attempt to test a large number of username and password combinations.

Throughput measurement. In throughput testing, we tested authentications-per-second (ApS) that are possible from a single desktop computer. Table 2.2 shows the throughput results for login testing. An interesting case was Expedia, which allowed ∼150 ApS when communicating with a single server IP and upwards of ∼600 ApS when using multiple server IPs during testing. The existence of multiple server IPs, either directly from the backend servers or CDN, played a role in the amplification of an attack. It is interesting to note that in the case of Expedia, different CDN IPs do not in fact allow amplification attacks.

App	ApS (Single-server-IP)	ApS (Multi-server-IP)	# of IPs found	CDN/Host
Ebay	∼ 77	∼100	2	Ebay
Expedia	∼150	∼600	20	Akamai/Expedia
SoundCloud	∼77	∼178	2	EdgeCast
We Heart It	∼83	∼215	5	SoftLayer/ThePlanet.com
Zappos	∼84	∼188	20	Akamai

Table 2.2: Throughput results for login testing.

We hypothesize that it is due to the fact that these CDNs still need to access the same set of backend servers which are the real bottleneck. To identify backend server IPs, we perform a step we call "domain name scanning" and successfully locate a non-CDN IP for "ftp.expedia.com". From this IP, we further scan the subnet and find 19 other IPs capable of performing authentication. By talking to these IPs directly, we are able to improve the throughput from 150 to 600.

Finally, we also obtain throughput results for 4 of the applications in sign up testing and their average throughput is around 90 to 240 ApS.

Client IP changing. During IP address testing, we find that 11 app-web pairs have client IP changing detection and associated security policy on the server side. The remaining 89 app-web pairs have no visible security policy. Among them there are 8 app-web pairs for which both the app and the website have the same behavior against IP changing. For the remaining 3 pairs, — Target, Twitch, Steam — the app and website have different behaviors where the website returns an access denied error for some IP address changes (in the case of Target and Twitch) or forces a logout for any change of the IP address (in the case of Steam) but the app allows changing client IP address frequently.

One main consequence is that when an app/website has no security policy against

App-Web	**App Cookies Expiration Time**	**Website Cookies Expiration Time**
AliExpress	several months	60 minutes
Amazon	several months	14 minutes
Best Buy	several months	10 minutes
Kohl's	several months	20 minutes
Newegg	several months	60 minutes
Walmart	several months	30 minutes

Table 2.3: Cookies expiration time.

IP changing, an attacker can perform HTTP session hijacking with stolen cookies more easily without worrying about what hosts and IP addresses to use in hijacking. For instance, Steam is a gaming client; it does have security protection in its websites. When a cookie is sent from a different IP, the website immediately invalidates the cookie and forces a logout. However, using the Steam app and the associated server interface, if the attacker can steal the cookie, he can impersonate the user from anywhere (i.e., any IP address).

Cookies. Cookies are commonly used for web services as well as mobile apps. In browsers, cookie management has evolved over the past few decades and gradually become more standardized and secure. However, on the mobile platform every app has the flexibility to choose or implement its own cookie management, i.e. cookie management is still far from being standardized.

We observe that many apps store their cookies unencrypted (47 apps among all 100 apps). An attacker can access the cookie more easily as compared to browsers on PCs. First, smartphones are smaller and more likely to be lost or stolen. Therefore, a simple dump of the storage can reveal the cookies (assuming no full-disk encryption). In contrast, in the case of browsers on PCs, cookies are often encrypted with secrets unknown to the

Library	Vulnerabilities	# of Apps	Example Vulnerable Apps(Version)(Installs)

libzip	DoS or possibly execute arbitrary code via a ZIP archive	13	com.djinnworks.StickmanBasketball(1.6)(10M+)
			com.djinnworks.RopeFly.lite(3.4)(10M+)

FFmpeg *	DoS or possibly have unspecified other impact	9	co.vine.android(5.14.0)(50M+)
			com.victoriassecret.vsaa(2.5.2)(1M+)

libxml2	DoS via a crafted XML document	8	com.avidionmedia.iGunHD(5.22)(10M+)
			com.pazugames.girlshairsalon(2.0)(1M+)

	Obtain sensitive information	5	com.pazugames.girlshairsalon(2.0)(1M+)
			com.flexymind.pclicker(1.0.5)(0.1M+)
			com.pazugames.cakeshopnew(1.0)(0.1M+)

	DoS or obtain sensitive information via crafted XML data	5	
	DoS via crafted XML data	5	
libcurl	Authenticate as other users via a request	1	sv.com.tigo.tigosports(6.0123)(0.1M+)

Table 2.4: Vulnerable libraries used by apps.

* FFmpeg includes 7 libraries:
libavutil, libavcodec, libavformat, libavdevice, libavfilter, libswscale, and libswresample.

attacker even if the attacker can gain physical access to the device. For instance, Windows password (used in Chrome) and master password (used in Firefox) are used to encrypt the cookies [112]. Second, if the device is connected to an infected PC (with adb shell enabled), any unprivileged malware on PC may be able to pull data from the phone. For instance, if the app is debuggable then with the help of *run-as* command, one can access the app data such as cookies. Even if the app is not debuggable, the app data can still be pulled from the device into a file with .ab(android backup) format [54].

We also report another type of important discrepancy — cookie expiration time. Here we focus on 18 shopping app-web pairs (a subset from the list of 100 pairs). We observe that app cookies remain valid for much longer time than web cookies. The cookie expiration time in all 18 shopping websites is around 3 hours on average, whereas it is several months in their app counterparts. The result is shown in Table 2.3. We find that 6 apps have cookie expiration time set to at least 1 month while their websites allow only minutes before the cookies expire. An attacker can easily use a stolen cookie for these apps and perform unwanted behavior such as making purchases. For instance, Amazon app appears to use cookies that never expire to give the best possible user experience. We confirmed that a user can make purchases after 1 year since the initial login in.

Vulnerable libraries. During vulnerable library testing, we find two apps (Vine and Victoria's Secret) use unpatched and vulnerable libraries from FFmpeg [7] framework, which motivates us to look at a larger sample of 6,400 top free apps in different categories. Table 2.4 summarizes our observation for vulnerable libraries with the number of apps using them. For example, an attacker can cause a DoS (crash the application) or possibly execute arbitrary code by supplying a crafted ZIP archive to an application using a vulnerable version of libzip library [11]. As we discussed before, javascript vulnerabilities are unlikely to cause damage to the device compared to app libraries, especially given the recent defences implemented on WebView [40].

2.5 Summary

We identified serious security related discrepancies between android apps and their corresponding website counterparts. By analyzing 100 mobile app-website pairs, we discovered that Android apps in general have weaker security policies in place as compared to their websites, likely due to either negligence or usability considerations on the smartphones. We responsibly disclosed all of our findings to the corresponding companies including Expedia who acknowledged and subsequently fixed the problem. The lesson learnt is that, for the same web service, even though their websites are generally built with good security measures, the mobile app counterparts often have weaker or non-existent security measures.

Chapter 3

Dynamic Slicing for Android

Dynamic program slicing is a very useful technique for a variety of tasks, from testing to debugging to security. However, prior slicing approaches have targeted traditional desktop/server platforms, and to the best of our knowledge there are no slicing techniques have been implemented in for smartphone mobile platforms such as Android. In this chapter, we present the challenges of slicing the event-based mobile apps and propose a technique (AndroidSlicer) to address the challenges effectively. Our technique combines a novel asynchronous slicing approach for modeling data and control dependences in the presence of callbacks with lightweight and precise instrumentation. We show our technique is capable of handling a wide array of inputs that Android supports without adding any noticeable overhead; making AndroidSlicer a very efficient and effective tool for variety of tasks.

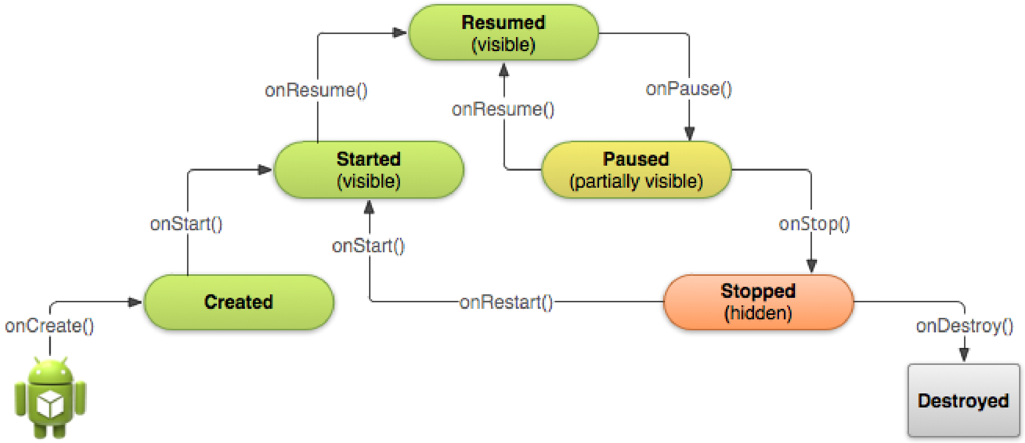


Figure 3.1: Android activity simplified lifecycle

3.1 Background

In this section, first we present a brief overview of Android platform and its event-based model, second we briefly introduce program slicing.

Android's event-based model. Android apps consist of components (*e.g.,* an app with a GUI consists of screens, named Activities[1]) and one or more entry points. Activities are the fundamental part of the platform's application model. Unlike traditional programming paradigms in which apps are launched with a *main()* method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle. Figure 3.1 shows the activity lifecycle in Android. Each Activity provides a core set of callbacks (*e.g.,* onCreate(), onStart(),onResume()) which are invoked by the system as an activity enters a new state. Hence, the even-driven model in apps dictates control flow of the program. Generally, a non-deterministic event

[1] The vast majority of Google Play apps consist of Activities. There are other component types such as Services, Content Providers, and Broadcast Receiver [20] but these are used much more sparsely.

can be a user action (*e.g.,* touch), a lifecycle event (*e.g., onPause()*), arrival of sensor data (*e.g.,* GPS), and inter- or intra-app messages. Non-deterministic manner of these callbacks challenges the slicing. We show the details of this challenge in 3.2.

Program Slicing. Dynamic program slicing, a class of dynamic analysis, was introduced by Korel and Laski [59] to assist programmers in debugging. The backward dynamic slice at instruction instance s with respect to *slicing criterion* $\langle t, s, value\rangle$ (where t is a timestamp) contains executed instructions that have a direct or indirect effect on *value*; more precisely, it is the transitive closure over dynamic data and control dependences in the PDG starting from the slicing criterion. The slicing criterion represents an analysis demand relevant to an application, *e.g.,* for debugging, the criterion means the instruction execution that causes a crash.

3.2 Android Slicing Challenges

We now show how the Android programming model/platform introduce challenges for constructing a dynamic slicer and hence we can not directly apply the traditional slicing techniques, and discuss how we have overcome these challenges.

3.2.1 Challenge 1: Low Overhead

Dynamic slicing (as with any dynamic analysis) on mobile platforms must not *interfere* with the execution of the app that is being analyzed. This requirement is *much more stringent* on mobile platforms than in traditional desktop/server programs, because mobile apps do not tolerate delays gracefully: we illustrate this with three examples.

App	**Original run** (s)	**AndroidSlicer run** (s)	**Pin run** (s)	**AndroidSlicer overhead** (%)
Indeed Job Search	15.8	17.1	Crashed	8
Geek	29.4	32.2	Crashed	9
Scanner Radio	29.5	30.9	Crashed	5
Daily Bible	23.9	24.2	Crashed	1
CheapOair	21.7	22.8	Crashed	5
Kmart	24.5	25.2	Crashed	3

Table 3.1: AndroidSlicer and Pin comparison.

First, even just attaching the standard dynamic analyzer Pin [68] to an Android app – a trivial operation on desktop/server – can have unacceptable overhead, or outright crash the app. To do the comparison with Pin, we instrumented 6 well-known apps using AndroidSlicer and Pin (for Pin we used a simple instrumenter that prints the number of dynamically executed instructions, basic blocks and threads in the app). Table 3.1 presents the results. Apps are sorted based on number of installs. We used Monkey with default settings to send the apps 5,000 UI events. Note that Pin instrumentation crashed all examined apps due to non-responsiveness[2] while AndroidSlicer instrumentation had a low overhead of 5% in average. Second, introducing delays in GUI event processing can alter the semantics of the event: an uninstrumented app running at full speed will interpret a sequence of GUI events as one long swipe, whereas its instrumented version running slower might interpret the sequence as two shorter swipes [41]. Third, harmful interference due to delays in GPS timing, or in event delivery and scheduling, can easily derail an execution [52].

Our approach. We address this challenge by optimizing register tracking at the AF/library boundary. First, in the runtime tracing phase, for a call into the AF/library

[2] https://developer.android.com/reference/java/util/concurrent/TimeoutException

we only track live registers, and only up to the boundary; upon exiting the AF/library we resume tracking registers. Second, in the static analysis phase we compute taint (source → sink) information to identify those methods that take values upward to the AF (sources) as well as those methods which return values downward to the app code (sinks). Finally, in the trace processing phase we instantiate the static taint information with the registers tracked into and out of the framework.

3.2.2 Challenge 2: High-throughput Wide-ranging Input

“Traditional” applications take their input mostly from files, the network, and the occasional keyboard or mouse event. In contrast, Android apps are touch- and sensor-oriented, receiving high-throughput, time-sensitive input from a wide range of sources. Typical per-second event rates are 70 for GPS, 54 for the camera, 386 for audio, and 250 for network [52]. A simple swipe gesture is 301 events per second [41]. Thus, we require low-overhead tracking of high-throughput multi-sourced input.

Our approach. Android employs AF-level event handlers for capturing external events. We achieve both scalability and precision by intercepting the registers at event processing boundary, as illustrated next. Swipes are series of touches, with the event handler *onFling(MotionEvent* e_1*, MotionEvent* e_2*, float velocityX, float velocityY).* We intercept the event by tracking the registers that hold the event handler parameters, i.e., e_1, e_2, *velocityX, velocityY*, and tagging them as *external inputs.* This approach has two advantages. First, register tracking is efficient, ensuring scalability. Second, being able to trace program behavior, *e.g.,* an app crash, to a particular external input via a backward slice allows developers to “find the needle in the haystack” and allows us to perform efficient and effective fault localization. Although our implementation targets Android, it is agnostic of the low-level OS layer.

3.2.3 Challenge 3: Finding Program Starting Points

Dynamic backward slicing requires traversing the execution back to the “beginning”. In traditional Java programs, this is straightforward: the slicer traverses the execution back to the beginning of the *main()* method. However, Android apps have multiple entry points. Moreover, each activity can be restarted, *e.g.,* due to screen rotation or when taking a call; when the activity is restarted, the *onCreate()* callback is invoked by the system. Hence we have to trace execution back to entry points or back across restarts. We illustrate this on a sample activity in Figure 3.2. Suppose we want to compute the slice with respect to variable name, using statement 7 as slicing start point. For this we need to traverse the set of reachable nodes in the PDG. The slice should contain statements {7, 6, 3, 2}. During execution, a configuration change (*e.g.,* screen rotate) will restart the activity. As a result, the *onCreate()* method is called to recreate the activity. Without a lifecycle-aware slicing approach, i.e., understanding that *onCreate()* is called upon restart, we would not be able to construct the correct slice (shown in Figure 3.2, right). Consequently, a traditional slicing approach cannot find the start point of slicing and would yield an empty slice rather than the correct slice {7, 6, 3, 2}. Therefore, the first challenge is to accurately find an apps entry points (including *onCreate()* in our example).

Our approach. To discover all entry points, we use a fixpoint approach. Note that Android apps must define an Activity which launches the app initially, known as the “home activity”. Therefore we start our analysis from the home activity and create pseudo-

```
1      public class AccountSetting
       extends Activity {
2   S   Account mAccount;
3   S   String name = "";
4       @Override
5       public void onCreate(Bundle
       savedInstanceState) {
6   S            mAccount = new Account();
7   S            name = name + mAccount.
       getName();
8       }}
```

Program

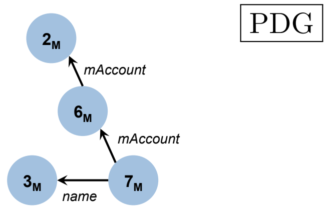


Figure 3.2: Program and its associated PDG. In the program: lines marked with an S denote the slice with respect to variable *name* on line 7. In the PDG: solid edges denote data dependences; graph nodes marked with an $_M$ denote nodes that would be missed by traditional slicing techniques. Labels on solid edges denote the variables which cause the data dependence.

entry points that correspond to callbacks. We borrow the concept of "pseudo entry point" from FlowDroid [22] but extend that approach with other necessary callbacks (i.e., callbacks accepting no parameters) to increase precision, according to the following strategy:

- Start with pseudo-entry points, including life-cycle callbacks, i.e., onCreate(), onResume(), onStart().
- Analyze the resulting callgraph using SOOT [104]. If the callgraph contains calls to AF events, such as button clicks, add the corresponding event-handling callbacks to the list of entry points. For outgoing calls, add the incoming callbacks, *e.g.,* for the outgoing call sendBroadcast() we add the corresponding callback onReceive() to the list of entry points.
- Continue this process until we converge to a fixpoint (no new callbacks to be added).

3.2.4 Challenge 4: Inter-App Communication.

Android relies heavily on inter-process communication (IPC). Originally designed to permit apps to access system resources in a controlled way, IPC mechanisms are also used for intra-app communication, *e.g.,* between two activities. The fundamental IPC mechanism is called Intent: using an intent, an activity can start another activity, or ask another app for a service, receiving the result via intents as well. For example, the Facebook app can send an Intent to the Camera app asking it to take a picture. there are two types of intents: implicit and explicit. An implicit intent starts any component that can handle the intended action, potentially in another app; because of this, an implicit intent does not name a specific destination component. An explicit intent specifies the destination component (a specific Activity instance) by name. Explicit intents are used intra-app to start a component that handles an action; we will describe the challenges introduced by explicit intents shortly, in next section. Implicit intents and consequently, inter-app communications, complicate slicing.

We illustrate this in Figure 3.3. The example shows the *GetContacts* activity that allows the user to pick a contact. An intent can launch an activity via the *startActivity* or *startActivityForResult* methods. Upon completion, Android calls the *onActivityResult* method with the request code that we have passed to *startActivityForResult* method (line 5 in the example). Without understanding the impact of inter-app intents, we would not be able to find complete slices. Assume we want to compute the slice with respect to variable name starting at statement 14. the resulting slice should contain statements {14, 9, 10, 12, 13, 8, 11, 5, 4}. However, using traditional slicing, we would not find the complete

```
1      public class GetContacts extends Activity {
2       @Override
3       public void onCreate(Bundle savedInstanceState) {
4  S      Intent i = new Intent(Intent.ACTION_PICK, Uri.
        parse("content://contacts"));
5  S      startActivityForResult(i, PICK_CONTACT_REQUEST)
        ;
6       }                                            Program
7       @Override
8  S    public void onActivityResult(int requestCode, int
         resultCode, Intent data) {
9  S      if (requestCode == PICK_CONTACT_REQUEST) {
10 S        if (resultCode == RESULT_OK) {
11 S          Uri contactData = data.getData();
12 S          Cursor c = getContentResolver().querty(
        contactData,null,null,null,null);
13 S          if (c.moveToFirst()) {
14 S            String name =c.getString(c.getColumnIdx(
        ContactsContract.Ctcs.DISP_NAME));
15 }}}}}
```

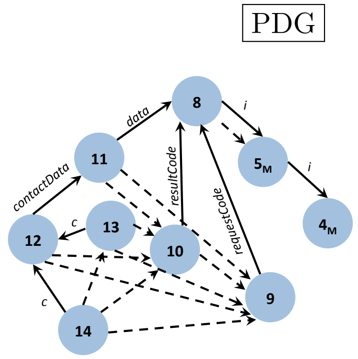


Figure 3.3: Program and its associated PDG. In the program: lines marked with an S denote the slice with respect to variable *name* on line 14. In the PDG: solid edges denote data dependences; dashed edges denote control dependences; graph nodes marked with an M denote nodes that would be missed by traditional slicing techniques. Labels on solid edges denote the variables which cause the data dependence.

slice because the technique only adds statements {14, 13, 12, 11, 10, 9} to the slice it will miss statements 4 and 5 for two main reasons. First, traditional slicing fails to pair *startActivityForResult()* with *onActivityResult()* – which are similar to a caller-callee – and thus it fails to reconstruct control flow to account for IPC. Second, note how we cross memory spaces into the Contacts app, hence we need to account for Androids sandboxing to be able to trace the initial (request) and result intents.

Our approach. We analyze app inputs and internal callbacks to detect intents and construct data dependence edges accordingly. In practice, IPC message objects (i.e., Intents) are processed by callbacks hence introduce asynchronous data dependences, which are naturally handled by our approach.

```
1     public class ActivityOne extends Activity
       {...
2  S   Intent i = new Intent(this, ActivityTwo.
       class);
3  S   i.putExtra("Value", "Some Value");
4  S   startActivity(i);
5     ...}
6     public class ActivityTwo extends Activity
       {...
7  S   Bundle extras = getIntent().getExtras();
8  S   String value = extras.getString("Value");
9     ...}
```

Program

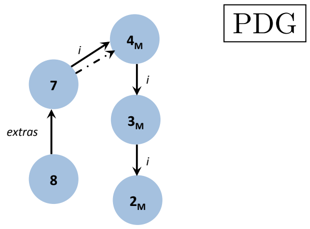


Figure 3.4: Program and its associated PDG. In the program: lines marked with an S denote the slice with respect to variable *value* on line 8. In the PDG: solid edges denote data dependences; graph nodes marked with an M denote nodes that would be missed by traditional slicing techniques. Labels on solid edges denote the variables which cause the data dependence.

3.2.5 Challenge 5: Intra-App Communication.

Explicit intents also complicate slicing, as shown in Figure 3.4. The example shows *ActivityOne* starting *ActivityTwo*; the message “Some Value” is passed via IPC mechanisms, the Bundle in this case. Let us assume we need to compute the slice with respect to variable value starting at statement 8. The dynamic slice should contain statements {8, 7, 4, 3, 2}. However, traditional slicing cannot find the precise slice because it does not account for intra-app communication. Specifically, the example uses Bundles *putExtra* and *getExtra* to pass the data between the two activities; the Bundle is a system component, so in this case the data flow is mediated by the system, and would elude a traditional slicer. Hence traditional slicing would not traverse statements {4, 3, 2} due to the missing dependences between the two activities and would yield slice {8, 7} which would be incorrect.

Our approach. To deal with challenges imposed by inter- and intra-app communication, we track callbacks and Android Framework APIs. For example if an activity calls another activity through Android APIs *startActivity* or *sendBroadcast* by passing an intent, we trace the receiver callback and the parameter referencing the intent. Hence such kinds of callbacks employing new communication impose dependences that must be handled.

To summarize, by recording callbacks and intents, AndroidSlicer captures inter-app and intra-app communication precisely, with no under- or over-approximation.

3.3 Algorithm Design

In this section, we describe AndroidSlicer's implementation. An overview is shown in Figure 3.5 : an offline instrumentation stage, followed by online trace collection, and a final offline computation stage. In the first stage, the app is instrumented to allow instruction tracing. Next, as the app executes, runtime traces are collected. Then we perform an on-demand static analysis to optimize trace processing, and then compute the PDG. Finally, we calculate slices for a given slicing criterion. We now discuss each phase.

3.3.1 Instrumentation

The purpose of this stage is three-fold: identify app entry points; construct method summaries; and add instruction/metadata tracing capabilities to the app.

Finding app entry points. As discussed in Section 3.2, identifying app entry points is a challenge that must be addressed in order to correctly compute backward dynamic slices. This is performed via a fixpoint computation, as described in Section 3.2. First, we use static analysis and create a "dummy" *main* method that systematically considers all system callbacks from a callback definition file.

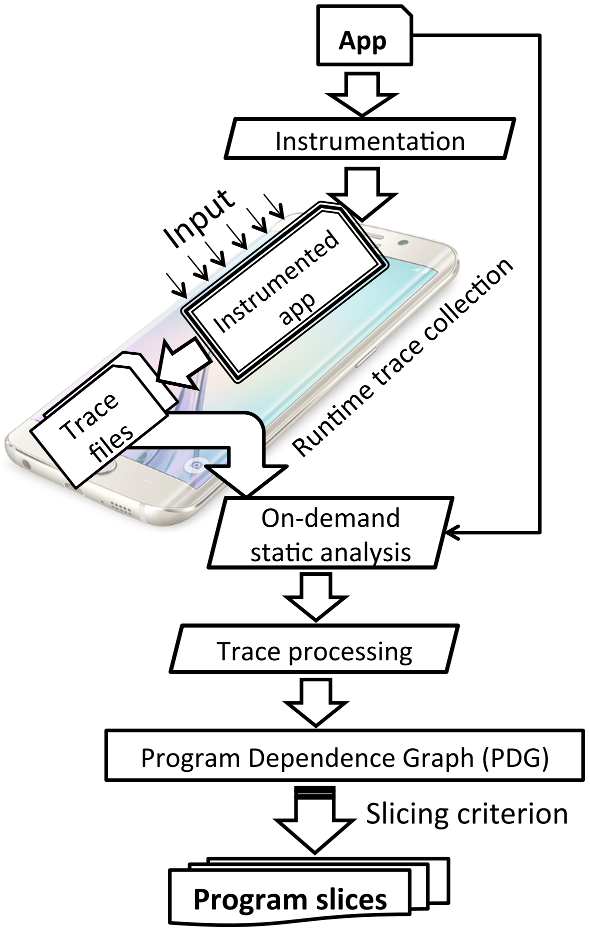


Figure 3.5: AndroidSlicer overview.

Constructing method summaries. Method summaries (which include in/out registers and method type) capture method information for the online trace collection phase. For this purpose, we first build a callgraph for each app class from the analyzed app entry points to create method summaries (i.e., in/out registers and method type). For each node in the callgraph (i.e., method) we add instrumentation tags that summarize that method. This instrumentation is an extended version of the method signature present in the Dexcode (Android bytecode); we save information for parameter registers and return value registers. We also detect callbacks at this time and add necessary information about input parameters.

We identify intents referenced through registers used as callback parameters and construct metadata such as caller information (i.e., name of the callback-generating and broadcasting the intent), as well as string properties associated with the intent's action filter. This information helps reveal *callers* and their *callees* during offline trace analysis.

Adding tracing instructions. We add tracing capabilities via Soot [104]. AndroidSlicer's instrumenter takes the app binary as input; the output is the instrumented app, which we run on the phone. To support tracing, we inject a new Dexcode instruction for every app instruction or callback routine. The trace format is described next.

3.3.2 Runtime Trace Collection

We collect the traces while the instrumented app is running on the phone. Traces have the following format:

```
trace_entry := <t, instruction_number_offset, instruction, [summary]>
summary := <type, invoked_method, parameter_registers,
return_registers, callback_parameter_registers,
intent_source, intent_action_filters>
```

Trace entries have the following semantics: `t` is the actual time the instruction was executed; `instruction_number_offset` is the instruction's relative line number in the printed dex code; `Summary` information is only used for method invocations; it contains the method's `type`, *e.g.,* an IPC or a non-IPC call, in/out register values, caller information, and where applicable, the action string (filter) associated with the intent.

3.3.3 On-demand Static Analysis

The PDG contains data and control dependence edges for regular nodes and supernodes (callbacks). To build the PDG efficiently, we conduct a post-run on-demand static analysis that uses the collected runtime information to narrow down the scope of the static analysis to only those app parts that have been exercised during the run. The advantage of this on-demand approach is that, instead of statically analyzing the whole program (which for Android apps raises scalability and precision issues), we only analyze those methods encountered during execution. The on-demand analysis phase performs several tasks.

3.3.4 Trace Processing and PDG Construction

With the static analysis information at hand, we analyze the application traces to generate the PDG of that particular execution. The PDG is built gradually via backward exploration of dependences, adding nodes and edges. Our prior static analysis produces two sets: (1) $StaticData_{si}$ – the set of static data dependence nodes for an instruction s_i, and (2) $StaticControl_{si}$ – the set of static control dependence nodes for an instruction s_i. As mentioned in Section 3.3, suffix t distinguishes between different occurrences of an instruction; the implementation uses a global counter for this purpose.

Sequential data dependence edges. For every occurrence of an instruction s_i, add a data dependence edge to the last executed occurrence of every instruction in its $StaticData_{si}$.

Sequential control dependence edges. For every occurrence of an instruction s_i, add a control dependence edge to the last executed occurrence in its $StaticControl_{si}$.

Asynchronous data dependence superedges. For every occurrence of a callback *callee* node, add a data dependence to the last occurrence of its *caller*. This information is revealed via static callback analysis. We also identify the instruction S_{ipct} that contains the actual IPC method call in the caller that passed the intent reference at time t. The callee receives the intent through one of its parameter registers v_{intent}. We then identify the last occurrence of the first instruction in *callee* at time t that uses v_{intent}. Let us name this S_{int}. We then add a data dependence $S_{ipct} \leftarrow_d S_{int}$.

Asynchronous control dependence superedges. If there is no data dependence between the corresponding supernodes from two consecutive activity contexts, i.e., callback *callee* and its *caller* (N_1 and N_2), we add a control dependence superedge $N_1 \leftarrow_c N_2$. Otherwise, we add a control dependence superedge $N_0 \leftarrow_c N_2$, where N_0 is the supernode N_1 is control-dependent on.

3.3.5 Generating Program Slices from the PDG

We now discuss our approach for generating slices given the PDG and a slicing criterion. The slicing criterion $\langle t, s_t, v_s \rangle$ represents the register v_s in instruction s at a particular timestamp t. Since an instruction is a regular node in the PDG we will use both terms interchangeably, i.e., s_t refers to both the exact instance of the instruction at time t and the PDG node. We maintain a workset T_s that holds the nodes yet-to-be-explored (akin to the working queue in Breadth-first search). The output OUT_{st} is the set of distinct nodes in the PDG we encounter while backward traversing from s_t to any of the app entry points affecting the value held in register v_s. We first traverse the edges in the PDG starting from s_t and create a dynamic data dependence table Def_n and a control

Algorithm 1 Dynamic program slicing

Input: PDG, slicing criterion $SliceCriterion = (t, s_t, v_s)$ **Output: set of nodes** OUT_{st}

```
procedure SLICE($SliceCriterion$)
    $T_s \leftarrow \{s_t\}$ // initialize workset $T_s$
    $OUT_{st} \leftarrow \{s_t\}$
    for all nodes $n$ that are in $T_s$ do
        calculate set $P_n = Def_n \cup Ctrl_n$
        for all nodes $n'$ in $P_n$ do
            if $n'$ is a supernode then
                Expand & extract the last regular node $n_r$
                Add $n_r$ to $Def_n$
            else if $n' \equiv n$ & $P_{n'} \equiv P_n$ then
                Merge $(n', n)$; remove $n'$ from $P_{n'}$
            else if previous occurrence of $n$ is in $P_{n'}$ & $P_{n'} \subset P_n$ then
                Merge $(n', n)$; remove $n'$ from $P_{n'}$
            else
                add $n'_i$ to $OUT_{st}$; add $n'$ to $T_s$; remove $n$ from $T_s$
            end if
        end for
    end for
end procedure
```

dependence table $Ctrl_n$ for each node n on paths to entry points. For each regular node n' in the set $P_n = Def_n \cup Ctrl_n$ we add n' to OUT_{st}. If n' is a supernode and $n' \in Def_n$ we expand n'. The expansion adds the last occurrence of the regular node n_r inside n' that broadcasts an intent to Def_n and recalculates P_n. Note that n_r passes the IPC reference (intents) in a register to the next supernode, and hence it should be included in the slice. Since the same instruction can appear multiple times because of different occurrences at different timestamps, this procedure adds nodes with the same instructions to the slice for each occurrence. This increases the size of the slice. To reduce the number of nodes in OUT_{st} we make two optimizations.

1. **Node merging.** Given different occurrences (i.e., at times t and t') of a

regular node (i.e., $n \equiv s_t, n' \equiv s_{t'}$) if $P_n = P_{n'}$ we merge n and n' into n_{merged}. For two different occurrences N and N' of the same supernode, we also apply merging: if N and N' have incoming or outgoing data dependence edges we expand the nodes and merge the individual instructions, i.e., regular nodes inside them; if N and N' are connected by control dependence edges only, we merge them.

2. Loop folding. In loops, for every new occurrence of a loop body instruction s, we will add a new node in the slice. But these nodes may point to the same set of data and control dependence in the PDG – they are different occurrences of s. To reduce these duplications, we merge two distinct nodes n and n' in the loop if the following conditions are met: (a) current occurrence of n' depends on the previous execution of n; (b) current occurrence of n depends on the current occurrence of n'; and (c) $P_{n'} \subset P_n$.

Let us call the new node created after the merge n_{merged}. Each time we find a different occurrence of the merged node we compute the set $P_{n_{merged}}$. Then we apply reduction to further reduce it to a single node.

3.3.6 Limitation

Since AndroidSlicer's instrumenter is based on Soot, it inherits Soot's static analysis size limitations, *e.g.,* we could not handle extremely large apps such as Facebook. Note that this is not a slicing limitation per se, but rather a static analysis one, and could be overcome with next-generation static analyzers.

3.4 Applications

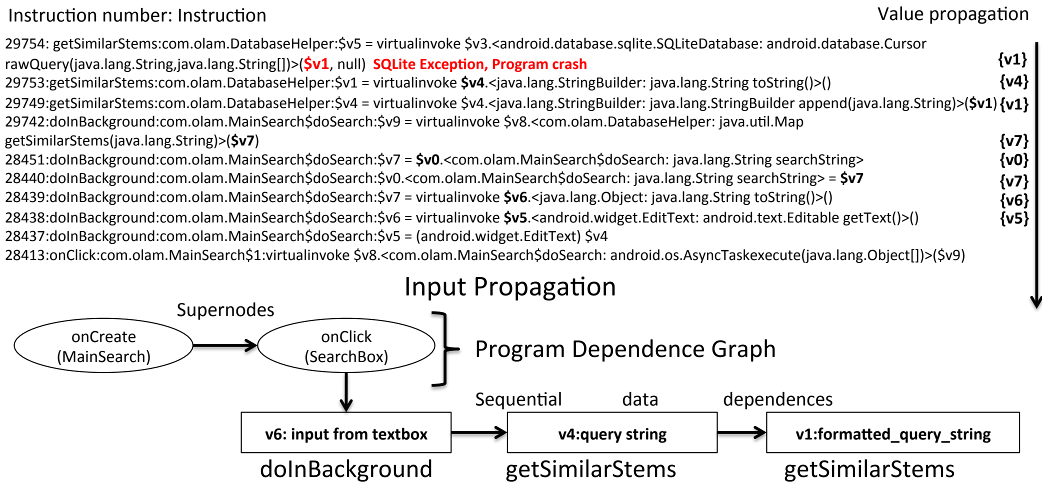
We describe three applications – failure-inducing input analysis, fault localization, and Regression test suite reduction – that leverage AndroidSlicer to facilitate debugging and testing in Android apps.

3.4.1 Failure-inducing Input Analysis

This analysis finds the input parts responsible for a crash or error. Note that unlike traditional programs where input propagation and control flow largely depend on program logic, in event-driven systems propagation depends on the *particular ordering of the callbacks associated with asynchronous events.* Leveraging our PDG, we can reconstruct the $input \rightarrow \ldots \rightarrow failure$ propagation path.

Problem statement. Let I be the set of app inputs $I_1, I_2, ...$ (*e.g.,* coming from GUI, network, or sensors) through registers $v_1, v_2, ...$ Let the faulty register be v_{err}, i.e., its value deviates from the expected value (including an incorrect numeric value, crash, or exception). Hence the analysis' input will be the tuple $\langle I, v_{err}, PDG \rangle$ while the output will be a sequence of registers $v_1, v_2, ..., v_n$ along with their callbacks $c_1, c_2, ..., c_m$.

Tracking input propagation. In the PDG, for every asynchronous callback, we can create an input propagation path by tracking the data dependence for the value of any register v_i. We determine whether the values propagated through registers are *influenced* by any of the app inputs I. This is particularly useful for identifying faults due to corrupted files or large sensor inputs (*e.g.,* a video stream).



```
29754: getSimilarStems:com.olam.DatabaseHelper:$v5 = virtualinvoke $v3.<android.database.sqlite.SQLiteDatabase: android.database.Cursor
rawQuery(java.lang.String,java.lang.String[])>($v1, null)  SQLite Exception, Program crash                                              {v1}
29753:getSimilarStems:com.olam.DatabaseHelper:$v1 = virtualinvoke $v4.<java.lang.StringBuilder: java.lang.String toString()>()          {v4}
29749:getSimilarStems:com.olam.DatabaseHelper:$v4 = virtualinvoke $v4.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>($v1) {v1}
29742:doInBackground:com.olam.MainSearch$doSearch:$v9 = virtualinvoke $v8.<com.olam.DatabaseHelper: java.util.Map
getSimilarStems(java.lang.String)>($v7)                                                                                                  {v7}
28451:doInBackground:com.olam.MainSearch$doSearch:$v7 = $v0.<com.olam.MainSearch$doSearch: java.lang.String searchString>              {v0}
28440:doInBackground:com.olam.MainSearch$doSearch:$v0.<com.olam.MainSearch$doSearch: java.lang.String searchString> = $v7              {v7}
28439:doInBackground:com.olam.MainSearch$doSearch:$v7 = virtualinvoke $v6.<java.lang.Object: java.lang.String toString()>()            {v6}
28438:doInBackground:com.olam.MainSearch$doSearch:$v6 = virtualinvoke $v5.<android.widget.EditText: android.text.Editable getText()>()  {v5}
28437:doInBackground:com.olam.MainSearch$doSearch:$v5 = (android.widget.EditText) $v4
28413:onClick:com.olam.MainSearch$1:virtualinvoke $v8.<com.olam.MainSearch$doSearch: android.os.AsyncTaskexecute(java.lang.Object[])>($v9)
```

Figure 3.6: Failure-inducing input analysis.

Example. We illustrate our analysis on an actual bug, due to a malformed SQL query, in Olam, a translator app.[3] The app takes an input word from a text box and translates it. In Figure 3.6 (top) we show the relevant part of the code: the number of distinct dynamic instances of an instruction (left), the actual instruction (center) and the value propagation through registers $v_1, v_2, ..., v_n$ along the PDG edges (right). In the method `getSimilarStems`, the app attempts to query the SQLite database, which generates an exception, resulting in a crash. The exception trace from the Android event log indicates that the query is ill-formed. The PDG (bottom left) points out the callback in which the exception was thrown: the `onClick` event associated with the search button in the `MainSearch` activity. We analyze the event inputs by following the data dependence edges backwards and see that the registers' values are pointing towards the input text from the textbox `editText`. We compute the slice using the faulty register reference as slicing criterion.

The execution slice is shown in Figure 3.6: we see that the ill-formatted string was

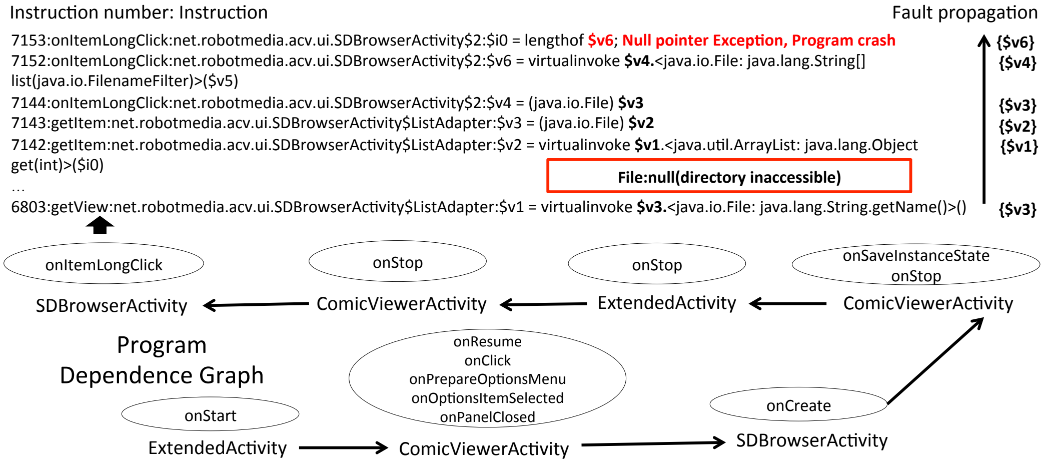
[3] `https://play.google.com/store/apps/details?id=com.olam`

stored in register v_1. Our approach back-propagates the value of v_1 to determine whether it was impacted by any part of the input. Back-propagation starts from the error location, i.e., instruction instance 29754. The value propagates to register v_5 which references the return value from `getText` invoked on an instance of v_4 that is pointing to the GUI control element `EditTextBox`. Our analysis ends by returning the register v_5 with the corresponding callback information. The second part of the figure shows the associated supernodes which reveal that the executed slices belong to the `MainSearch:onClick` callback. The failure-inducing input was thus essentially identified analyzing a much smaller set of instructions, and more importantly, in the presence of non-deterministic callback orders.

3.4.2 Fault Localization

This analysis helps detect and identify the location of a fault in an app. For sequential programs, fault localization is less challenging in the sense that it does not need to deal with the non-determinism imposed by asynchronous events. Android apps are not only event-driven but also can accept inputs at any point of the execution through sensors, files, and various forms of user interactions. For this reason, fault localization on Android can be particularly challenging for developers.

Problem statement. The input to the analysis will be the application trace, and the register v_{err} holding the faulty value in a specific occurrence of an instruction. The output this time will be the sequence of instructions $s_1, s_2, ..., s_n$ that define and propagate the value referenced in v_{err}.



```
7153:onItemLongClick:net.robotmedia.acv.ui.SDBrowserActivity$2:$i0 = lengthof $v6; Null pointer Exception, Program crash
7152:onItemLongClick:net.robotmedia.acv.ui.SDBrowserActivity$2:$v6 = virtualinvoke $v4.<java.io.File: java.lang.String[]
list(java.io.FilenameFilter)>($v5)
7144:onItemLongClick:net.robotmedia.acv.ui.SDBrowserActivity$2:$v4 = (java.io.File) $v3
7143:getItem:net.robotmedia.acv.ui.SDBrowserActivity$ListAdapter:$v3 = (java.io.File) $v2
7142:getItem:net.robotmedia.acv.ui.SDBrowserActivity$ListAdapter:$v2 = virtualinvoke $v1.<java.util.ArrayList: java.lang.Object
get(int)>($i0)
...
6803:getView:net.robotmedia.acv.ui.SDBrowserActivity$ListAdapter:$v1 = virtualinvoke $v3.<java.io.File: java.lang.String.getName()>()
```

Figure 3.7: Fault localization.

Tracking fault propagation. Our slicing approach aids fault localization as follows. Given a fault during an execution, we determine the faulty value reference inside a register v_{err} by mapping the Android event log to our execution trace. Then we compute the execution slice for v_{err} by back propagating through the execution slice. While we traverse the PDG backwards, we consider asynchronous callbacks and their input parameters if they have a direct data or control dependence to the final value of v_{err}. This way, we can both handle the non-determinism of the events and also support the random inputs from internal and external sources.

Example. We illustrate our approach on a real bug in the comic book viewing app ACV.[4] Figure 3.7 shows the generated dependences for the faulty execution. The bug causes a crash when the user opens the file explorer to choose a comic book. If the user long-taps on an inaccessible directory, the app crashes with a `null` pointer exception. From Figure 3.7 we can see that the object reference stored in register v_6 at instruction 7153 was the primary cause of the error. The corresponding callback is revealed to be `onItemLongClick`

[4] `https://play.google.com/store/apps/details?id=net.androidcomics.acv`

in activity `SDBrowserActivity`. Our analysis tracks back the object reference in v_6, reaching instruction 6803. Here we can see a file system API invocation (`java.io.File.getName()`) that attempts to return a filename, but fails because the file's directory is inaccessible. Our value propagation ends here, revealing the source of the error. We return the set of instructions $\{6803, ..., 7142, 7143, 7144, 7152, 7153\}$, and the traversed PDG nodes. For simplicity, we only show the data dependence edges and relevant parts of the slices. Our approach then back-propagates through the PDG according to the execution slice to localize the fault (for presentation simplicity we have combined consecutive supernodes in the same activity into a single node).

3.4.3 Regression Test Suite Reduction

Regression testing validates that changes introduced in a new app version do not "break" features that worked in the previous version. However, re-running the previous version's entire test suite on the new version is time-consuming and inefficient. Prior work [14, 45] has shown that slicing reduces the number of test cases that have to be rerun during regression testing (though for traditional apps).

Problem statement. Given two app versions (V_1 and V_2), and a test suite T_1 (set of test cases) that has been run on V_1, find T_2, the minimal subset of T_1, that needs to be rerun on V_2 to ensure that V_2 preserves V_1's functionality.

Test case selection. Agrawal et al. [14] used dynamic slicing to find T_2 as follows: given a program, its test cases, and slices for test cases, after the program is modified, rerun only those test cases whose slices contain a modified statement. This reduces the test suite because only a subset of program statements (the statements in the slice) have an effect on the slicing start point (program output, in their approach [14]). However, this technique can be unsound, because it only considers whether a statement has been modified, not *how it has been modified*. When the changed instructions affect predicates leading to an asynchronous control dependence, missed control dependences will lead to potentially missing some test cases. Our approach considers such dependences to maintain soundness.

3.5 Evaluation

We first evaluate AndroidSlicer's core slicing approach; next, we evaluate it on the three applications from Section 3.4.

Environment. An LG Nexus 5 phone (Android version 5.1.1, Linux kernel version 3.4.0, 2.3 GHz) for online and an Intel Core i7-4770 CPU (3.4 GHz, 24 GB RAM, 64-bit Ubuntu 14.04 kernel version 4.4.0) for offline processing.

3.5.1 Core Slicing

App dataset. We ran AndroidSlicer on 60 apps selected from Google Play, the official Android app store. The apps were selected from a wide range of categories (shopping, entertainment, communication, etc.) and with various bytecode sizes to ensure diversity in tested apps. In Table 3.2, we present detailed results for all apps sorted by number of installs. We summarize the findings (min/median/max) in the last three rows. The second column shows the app's bytecode size (median size of 1,485 KB). The third column shows app popularity (number of installs, *in thousands*, per Google Play as of August 2018). 28 apps had more than one million installs (median popularity 500,000 - 1,000,000 installs).

Generating inputs and slicing criteria. We used Monkey [21] to send the app 1,000 UI events and then collected traces. To measure 's runtime overhead, same event sequence was used in instrumented and uninstrumented runs. For slicing criteria, variables were selected to cover all types of registers (local variables, parameters, fields) from a variety of instructions (static invokes, virtual invokes, conditions, method returns), allowing us to draw meaningful conclusions about slicing effectiveness and efficiency.

Correctness. We manually analyzed 10 out of the 60 apps to evaluate AndroidSlicer's correctness. The manual analysis effort in some apps can be too high, because of the large number of instructions and dependences (*e.g.,* in the *Twitch* app, there are 5,969 instructions in the slice and 9,429 dependences). Therefore, we picked 10 apps whose traces were smaller so we could verify them manually with a reasonable amount of effort. We decompiled each app to get the Java bytecode, and manually computed the slices from the slicing criterion. The manually-computed slices were then compared with AndroidSlicer's; we confirmed that slice computation is correct, with no instruction being incorrectly added or omitted.

Effectiveness. Table 3.2 demonstrates that AndroidSlicer is effective. The "Instructions Executed" column shows the total number of instructions executed during the entire run. The median number of instructions is 14,491. If the programmer has to analyze these, the analysis task will be challenging. AndroidSlicer reduces the number of instructions to be analyzed to *44, i.e., 0.3%* (column "Instructions In slice"). The median number of dependences to be analyzed, data and control, is not much larger, 63, (column "CD+DD"). The next column shows the number of callback events fired during the run.

App	**Dex**	**Installs**	**Instructions**		**CD**	**Callback**	**Time (seconds)**				**Over-**
	code		**Executed**	**In slice**	**+DD**	**events**	**Stage 1**	**Original**	**Stage 2**	**Stage 3**	**head**
	size (KB)	**(thousands)**					**Instrumentation**	**run**	**Instrumented run**	**Slicing**	**(%)**
Twitter	50688	500000-1000000	107847	559	563	557	293.5	233.4	236.5	40.3	1
*Indeed Job Search	2457	50000-100000	21752	246	264	235	24.0	222.2	234.3	12.1	5
Geek	12902	10000-50000	152640	1422	1423	1420	116.3	215.2	221	84.9	2
GroupMe	14438	10000-50000	45876	471	476	467	81.1	205.4	212.8	45.4	3
ROM Manager	7270	10000-50000	141333	181	184	178	82.1	233.7	236.9	56.3	1
*Scanner Radio	9216	10000-50000	83190	745	750	741	120.6	231	238.2	51.6	3
Twitch	30105	10000-50000	2025505	5969	5978	5965	144.2	260.3	284.9	103.0	9
Weather by WeatherBug	18841	10000-50000	235773	355	356	353	141.6	242.3	245.7	68.6	1
*AHABER	1228	1000-5000	166	17	19	1	17.0	237.4	240	9.2	1
Apps Organizer	850	1000-5000	28685	15	22	12	7.7	241.6	262.5	5.9	8
BankID sakerhetsapp	1638	1000-5000	25833	19	24	1	22.4	233.1	237.4	13.2	1
Energy Saver	9011	1000-5000	175612	150	198	139	65.9	235.6	242.3	54.2	2
Hypnotic Spiral	47	1000-5000	15356	270	280	1	3.7	238.5	240.3	2.9	0
Idiotizer Pro	299	1000-5000	2426	32	33	30	3.9	248.8	254.7	8.2	2
iTelHybridDialer	1433	1000-5000	1899	19	23	18	17.1	214.8	221.5	8.7	3
Johnny's web	621	1000-5000	246	24	27	23	13.2	257.7	262.7	9.4	1
Turkish English Translator	1536	1000-5000	23525	65	72	63	19.2	227	260.3	24.9	14
*XFINITY Home	2150	1000-5000	1567	23	28	19	10.7	219.9	223.4	14.9	1
Contactor Select	373	500-1000	1362	6	6	4	2.5	223.9	227.4	3.6	1
Notepad for Android	2867	500-1000	17	4	8	2	30.6	233.5	237.8	7.2	1
Phone2Phone Internet Call.	764	500-1000	10487	800	952	748	14.1	229.6	253.1	8.2	10
The Art of War E-Book	404	500-1000	16855	5	8	4	6.5	232.1	238.5	4.4	2
Weight Diary	1740	500-1000	24017	6	9	5	34.0	240.5	244.6	14.8	1
CallTrack	75	100-500	114	4	5	2	3.6	227.6	235.3	4.2	3
Droid Splitter	467	100-500	20322	48	54	14	3.3	220.5	223	4.0	1
Element53 Lite	411	100-500	31399	80	88	75	9.7	237.8	255.4	8.0	7
Ethiopian Calendar	290	100-500	1671	105	124	92	2.3	228.5	233.4	3.2	2
Event Planner	2252	100-500	15846	186	202	184	21.0	234	256.1	12.2	9
Learn Advertising&Mktng.	2150	100-500	13626	72	77	70	16.6	219.6	238.5	21.9	8
NewsBook News Reader	2150	100-500	6935	24	39	20	41.3	228.3	243.5	24.7	6
Noticias Caracol	1331	100-500	1794	42	43	40	21.5	223.6	228.7	6.5	2
Öffnungszeiten Österreich	1638	100-500	7933	56	65	55	23.5	240.8	241.4	11.4	0
Out Call Blocker	440	100-500	1546	3	6	2	7.8	221.7	234.5	5.0	5
PDD Rus	3072	100-500	3195	51	62	50	30.5	228.6	241.4	5.9	5
Scrollable News Widget	759	100-500	1489	28	34	19	5.7	241.2	260.2	5.2	7
Time-Lapse - Lite	331	100-500	2274	8	10	5	7.2	234.6	236.7	5.1	0
backport.android.bluetooth	143	50-100	460	25	29	23	4.3	231	232.4	6.4	0
Digital Tasbeeh Counter	417	50-100	28911	3	4	2	7.2	219.4	221.5	6.5	0
Glasgow	1433	50-100	1144	75	79	60	3.7	240.6	256.7	3.5	6
*Mirrord Picture Reflection	1331	50-100	16938	73	82	65	17.3	238.5	249	13.3	4
Power writer	482	50-100	33383	4	5	2	6.9	227.3	241.7	22.5	6
*Alo	1740	10-50	129	5	8	1	21.9	224.3	236.5	33.8	5
*Australian Postcode Search	1024	10-50	3350	13	18	8	3.7	260.7	266.3	3.7	2
Fail Log	233	10-50	135	13	17	9	3.6	230.2	234.8	6.9	1
Got-IT! Free	1843	10-50	19664	240	256	237	5.7	253.9	255.6	13.8	0
Grid Size Free File Manager	296	10-50	3830	9	13	7	9.6	231	237.5	6.4	2
Pad - a simple notepad	91	10-50	35	2	2	1	3.6	217.1	235.6	2.0	8
Quickcopy	193	10-50	1360	19	23	17	4.3	229.7	237.8	4.1	3
StayOnTask	987	10-50	2297	24	26	23	5.3	238.7	269.2	6.6	12
TagNote	160	10-50	935	17	26	13	4.0	231.8	246.8	7.2	6
*Time Tracker	1433	10-50	57	5	6	4	20.9	225.7	233.6	24.3	3
*ToDoid	976	10-50	5518	93	104	89	5.5	220.3	229.7	6.3	4
TPV-POS Haird. Peluq.	2150	10-50	1019	5	6	4	10.3	226.9	238.8	7.9	5
*Upvise Projects	968	10-50	11903	11	26	22	18.9	229.5	247.7	6.5	7
Vremenska napoved	3174	10-50	6131	17	22	14	29.5	230.3	251.5	12.4	9
Bundlr	3174	5-10	3159	101	120	99	19.2	241.6	246.1	11.6	1
HNotes	671	5-10	82937	6	8	3	3.7	236	247.7	27.1	4
HP Designjet ePrint & Share	2560	-	2377	27	32	25	23.8	238.6	261.1	16.4	9
Update your phone	182	-	10	3	3	1	9.0	224.4	231.3	6.5	3
Yandex.Auto	1536	-	2109	5	7	4	5.9	246.7	272.1	6.4	10
Across *min*	*47*	*10-50*	*10*	*2*	*3*	*1*	*2.3*	*205.4*	*212.8*	*2.0*	*0*
all 60 *median*	*1,485*	*500–1000*	*14,491*	*44*	*63*	*23*	*19.1*	*229.85*	*239.7*	*11.9*	*4*
apps *max*	*50,688*	*500,000-1,000,000*	*2,025,505*	*5,969*	*5,978*	*5,965*	*293.5*	*260.7*	*284.9*	*103.0*	*14*

Table 3.2: AndroidSlicer evaluation: core slicing results.

Efficiency. The remaining columns ("Time" and "Overhead") show that AndroidSlicer is efficient. We show the time for each processing stage. Stage 1 (instrumentation), typically takes just 19.1 seconds, and at most 293.5 seconds for the 50.6 MB Twitter app. The "Original run" column shows the time it took to run the uninstrumented app – typically 229.85 seconds, and at most 260.7 seconds. Column "Stage 2 Instrumented run" shows the time it took to run the instrumented app, while collecting traces. The typical run time increases to 239.7 seconds. The "Stage 3 Slicing" column shows post-processing time, i.e., computing slices from traces, including on-demand static analysis; this time is low, typically just 11.9 seconds, and at most 103 seconds. The "Overhead" column shows the percentage overhead between the instrumented and uninstrumented runs; the typical figure is 4% which is very low not only for dependence tracking, but for any dynamic analysis in general. Furthermore, our instrumentation strategy does not require monitoring the app or attaching the app to a third-party module – this allows the app to run at its native speed. We emphasize that AndroidSlicer's *low overhead is key* to its usability, because Android apps are timing-sensitive (Section 3.2).

3.5.2 Failure-inducing Input Analysis

We evaluated this application on real bugs in 6 sizable apps (Table 3.3) by reproducing the bug traces. Our failure-inducing input analysis is very effective at isolating instructions and dependences of interest – the number of executed instructions varies from 320 to 182,527, while slices contain just 16–57 instructions. The CD and DD numbers are also low: 18–73.

App	Dex	Installs	Instructions		Log	CD+	Call-	Time (seconds)				Over-
	code		Executed	In slice	size	DD	back	Stage 1	Original	Stage 2	Stage 3	head
	size (KB)	(thousands)			(KB)		events	Instrumentation	run	Instrumented run	Slicing	(%)
Etsy	5,400	10,000–50,000	182,527	19	20,623	24	9	94	8.7	10.4	129.2	19
K-9 Mail	1,700	5,000–10,000	13,042	30	1,937	34	16	89.1	107.4	125.3	58.8	16
AnyPlayer M. Player	780	100–500	26,936	16	3,714	18	11	21.9	7.6	7.8	17.2	2
Olam M. Dictionary	651	100–500	31,599	57	3,802	73	22	17.3	46.7	50.1	19.4	3
Etsy	5,400	10,000–50,000	182,527	19	20,623	24	9	94	8.7	10.4	129.2	19
K-9 Mail	1,700	5,000–10,000	13,042	30	1,937	34	16	89.1	107.4	125.3	58.8	16
AnyPlayer M. Player	780	100–500	26,936	16	3,714	18	11	21.9	7.6	7.8	17.2	2
Olam M. Dictionary	651	100–500	31,599	57	3,802	73	22	17.3	46.7	50.1	19.4	3
¿¿¿¿¿¿¿ .r2562VuDroid	475.5	100–500	320	21	38	27	20	8.7	6.2	6.7	6.4	8
Slideshow	3,700	10–50	68,013	43	9,918	52	22	52.6	7.2	8.1	28.9	12

Table 3.3: AndroidSlicer evaluation: Failure-inducing input analysis.

App	Dex	Installs	Instructions		Log	CD+	Call-	Time (seconds)				Over-
	code		Executed	In slice	size	DD	back	Stage 1	Original	Stage 2	Stage 3	head
	size (KB)	(thousands)			(KB)		events	Instrumentation	run	Instrumented run	Slicing	(%)
SoundCloud	516.3	100,000–500,000	9,590	128	1,910	173	62	41.7	63.5	71.6	32.7	9
Notepad	44.2	10,000–50,000	2,366	15	343	17	6	4.5	36.5	41	9.1	12
A Comic Viewer	569.1	1,000–5,000	12,679	18	2,007	24	13	26.7	52.6	61.3	18.7	16
AnkiDroid Flashcards	804.6	1,000–5,000	27,164	32	3,722	38	27	87.6	17.3	19.5	28.7	12
APV PDF Viewer	52.9	1,000–5,000	24,672	67	3,436	79	45	11	10.3	11.2	27.2	8
NPR News	285.1	1,000–5,000	45,298	239	5,473	327	107	28.7	49.3	52.7	42.5	6
Document Viewer	3,900	500–1000	5,451	8	854	11	2	11.2	34	36.1	9	6

Table 3.4: AndroidSlicer evaluation: Fault localization.

3.5.3 Fault Localization

We evaluated our approach on 7 apps. Table 3.4 shows the results. Note how fault localization is effective at reducing the number of instructions to be examined from thousands down to several dozen. SoundCloud and NPR News have large slices due to intense network activity and background services (audio playback), which increase the callback count substantially.

3.5.4 Regression Test Suite Reduction

We evaluated our reduction technique on 5 apps. For each app, we considered two versions V_1 and V_2 and ran a test suite T_1 that consisted of 200 test cases; on average, the suite achieved 62% method coverage. Next, we used to compute the reduced test suite as described in Section 3.3.

App	**Dex code size V_1–V_2 (KB)**	**Installs**	**Test suite size**	**Covered methods (%)**	**Instructions**		**Reduced test suite**
					V_1	**V_2**	
Mileage	443.8-471.3	500-1,000	200	66	28,252	36,531	22
Book Catalogue	444.9-445.4	100-500	200	69	28,352	28,450	7
Diary	125.5-129.8	100-500	200	53	4,591	4,842	47
Root Verifier	462.9-1700	100-500	200	58	23,482	83,170	5
Traccar Client	49.4-51.6	50-100	200	66	1,833	1,937	8

Table 3.5: AndroidSlicer evaluation: Regression testing.

Table 3.5 shows the results: the bytecode sizes for V_1 and V_2, the number of installs, the coverage attained by T_1 on V_1, and the instructions executed when testing V_1 and V_2, respectively. The last column shows the size of T_2. Notice how our approach is very effective at reducing the test suite size from 200 test cases down to 5–47 test cases.

3.6 Related Work

Slicing event-based programs has been investigated for Web applications [73, 103, 87] written in HTML, PHP, and JavaScript. These approaches record traces through a browser plugin [73] and construct the UI model to generate the event nodes. While both Web and Android apps are event-based, their slicing approaches differ significantly. First, Android apps life-cycle cause apps to run in different scopes (i.e., activity, app, system), and handle different sets of requests (launch another activity, respond to an action, pass data). In contrast, Web apps have different build phases: UI building phase (HTML nodes) and event-handling phase (JavaScript nodes). Second, Web app slicing tool (*e.g.,* as a browser plugin) does not require low-overhead like in Android. Third, Android requires IPC tracking; that is not the case for Web.

Traditional program slicing of Java bytecode has only targeted single-entry sequential Java programs [109, 107, 2, 97]. Zhou et al. [122] and Zeng et al. [116] have used bytecode slicing for Android apps, but to achieve entirely different goals: mining sensitive credentials inside the app and generating low-level equivalent C code. They create slices at bytecode level and consider data dependences only; this makes the approach imprecise as there is no tracking of code dependences or accounting for many Android features (*e.g.,* callbacks, IPC, input from sensors).

Compared to Agrawal and Horgan's slicing for traditional programs [13], we add support for Android's intricacies, node merging for control dependence edges, dealing with slicing in the presence of restarts as well as asynchronous callback invocation. We support loop folding for regular nodes inside the supernodes. Slicing multithreaded programs is tangentially related work, where slicing was used to debug multithreaded C programs [118, 99, 98, 110, 108] — this setup differs greatly from ours.

Hoffmann et. al. developed SAAF [48], a static slicing framework for Android apps. A static slicing framework such as SAAF would not be sufficient to achieve our goals as it does not consider myriad aspects, from late binding to the highly dynamic event order in real-world Android apps.

3.7 Summary

We presented AndroidSlicer, a novel slicing approach and tool for Android that addresses challenges of event-based model and unique traits of the platform. Our asynchronous slicing approach that is precise yet low-overhead, overcomes the challenges.

Experiments on real Android apps show that AndroidSlicer is effective and efficient. We evaluated three slicing applications that reveal crashing program inputs, help locate faults, and reduce the regression test suite. In the future we plan to investigate forward slicing [27, 49] and language-agnostic slicing that would permit slicing apps containing code in different programming languages [28].

Chapter 4

Applications of Dynamic Slicing in Android

In this chapter we explore two new problems (bug and vulnerability) and show how we use the dynamic slicing technique introduced in the previous chapter to identify these two types of new bug/vulnerability. In the first problem, we address the issue of missing progress indicators in mobile apps. We present a novel *semantic* approach based on slicing for automatic detection of missing progress indicators for long-running operations such as network communications in mobile apps. In the second problem, we briefly show a new type of vulnerability in unique device identification techniques used in mobile apps which can lead to financial loss.

4.1 Missing Progress Indicators

A standard tenet of user interface (UI) design is to maintain “visibility of system status” [81]: the user should always be made aware of what work is currently being performed by the software. In particular, when a program is performing some long-running operation in response to a user action, a *progress indicator* should be displayed to indicate that the operation is ongoing. Missing progress indicators can lead to user confusion and frustration, as noted by Nielsen [80]:

> ”Progress indicators have three main advantages: They reassure the user that the system has not crashed but is working on his or her problem; they indicate approximately how long the user can be expected to wait, thus allowing the user to do other activities during long waits; and they finally provide something for the user to look at, thus making the wait less painful.”

User interface guidelines frequently emphasize the importance of consistent use of progress indicators [5, 6, 91] as one of the main UI design principles.

Hence, we present a new technique for *automatically* detecting cases where an application is missing an appropriate progress indicator. We focus on the scenario where some user interaction leads to a long-running operation, and the output of that operation is required to render a final result. This scenario is extremely common in networked mobile and web applications, as any network request can potentially be long running, depending on network conditions and the amount of data requested.

Recent work by Kang et al. [58] presents a technique for detecting unresponsive Android UIs, which are often caused by a missing progress indicator. Their technique works by monitoring app execution at a system level and detecting cases where after a user input, there are no UI updates of any kind for some given time period. While this approach finds

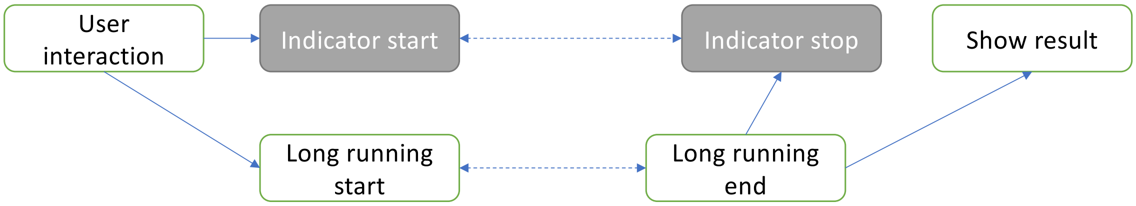


Figure 4.1: High-level illustration of relevant dependencies for correct use of a progress indicator.

many real issues, it suffers from having no way to distinguish different types of UI updates. Their approach may treat a non-progress-related UI update as evidence of a responsive UI, leading to missed bugs. (See Section 4.1.1 and Section 4.1.6 for further discussion.)

Unlike Kang et al. [58] work, we present a novel *semantic* approach to detecting missing progress indicators based on *program dependencies* [38]. Figure 4.1 gives a high-level illustration of relevant dependencies and matchings for correct progress indicator usage. The white nodes represent the main logic processing the user request: a user interaction triggers the start of a long running operation, and the result of the operation can only be shown once it has completed. The grey nodes indicate desired indicator behavior: the user interaction should also trigger the display of a progress indicator, which should be stopped after the long-running operation has ended. The dashed edges reflect the natural pairing between the start and stop of the long-running operation and display of the progress indicator.

The solid edges in Figure 4.1 represent (forward) program dependencies, which can be control or data dependencies depending on the scenario. With this formulation, missing progress indicators can be detected via the following two high-level steps:

1. Find sequences of program operations matching the white nodes and their dependencies (user interaction, start of long-running operation, etc.).

2. For each such sequence from step 1, ensure that there are interleaved progress indicator operations with the corresponding dependencies. If such indicator operations are absent, report a bug.

This approach requires knowing which program operations map to the corresponding nodes in Figure 4.1; e.g., we must know which operations start and stop progress indicators. Hence, some level of manual semantic modeling is required. However, once these semantics are captured, the approach has a number of desirable properties:

- More complete bug detection is possible compared to previous work [58], since the approach can distinguish progress indicator operations from other unrelated UI updates.

- When bugs are discovered, they can easily be mapped to source constructs when reporting to the developer, easing understandability and bug fixing.

- Decades of work from the research community on computing program dependencies [38] and program slices [102] can be applied to the problem.

We instantiate the above approach in a tool ProgressDroid for finding missing progress indicators in Android applications. Figure 4.2 gives an overview of the different components of ProgressDroid. As inputs, ProgressDroid requires an app and also *API mappings* detailing how to map API calls to the high-level operations shown in Figure 4.1. This mapping is mostly app-independent, but it can also accommodate app-specific APIs, e.g., to handle custom progress indicator libraries. ProgressDroid leverages the recently-released AndroidSlicer [24] to compute dynamic slices for apps. For Android, de-

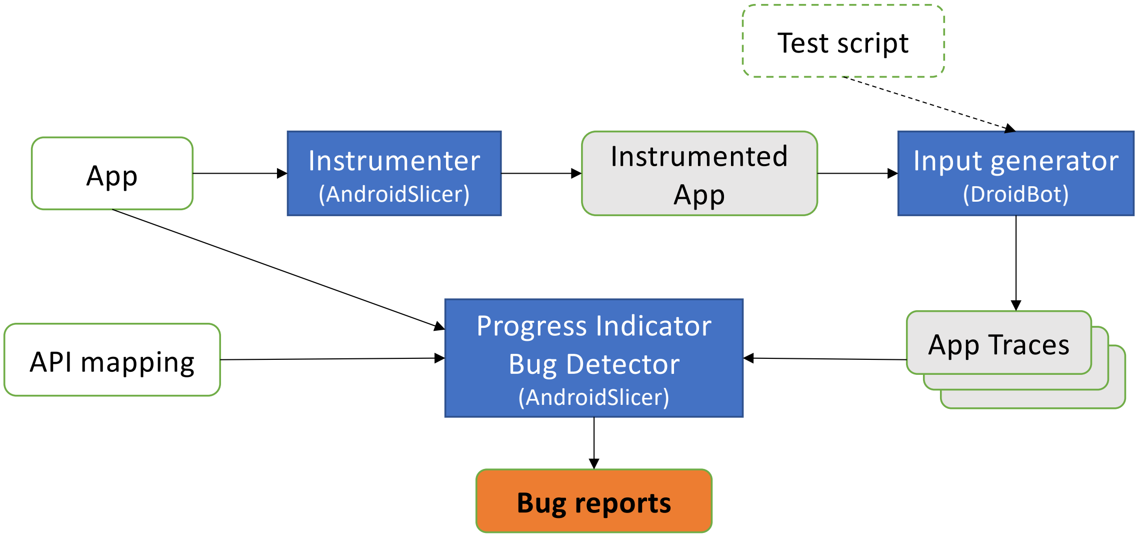


Figure 4.2: An overview of the different components of ProgressDroid.

pendencies related to user interactions, UI operations, and long-running network requests often cut across threads and asynchronous operations, and AndroidSlicer has built-in support for tracking such dependencies. ProgressDroid also uses DroidBot [64] to automatically exercise app behaviors, optionally aided by a custom test script.

Developing a technique and tool capable of detecting progress indicator bugs in real-world apps required overcoming three key challenges. First, a naïve use of program slices to relate long-running operations to user interactions leads to conflation of chained operations. Section 4.1.2 discusses this issue and introduces the notion of *immediate dependence* to capture the problem precisely, and Section 4.1.4 gives an algorithm for precise bug finding based on dynamic slicing. Second, we found that the semantic dependencies in AndroidSlicer [24] did not cover some key cases required for handling our use case, and we had to add support for these cases without compromising performance (see Section 4.1.5). Finally, we found that computing slices with AndroidSlicer was the most expensive part

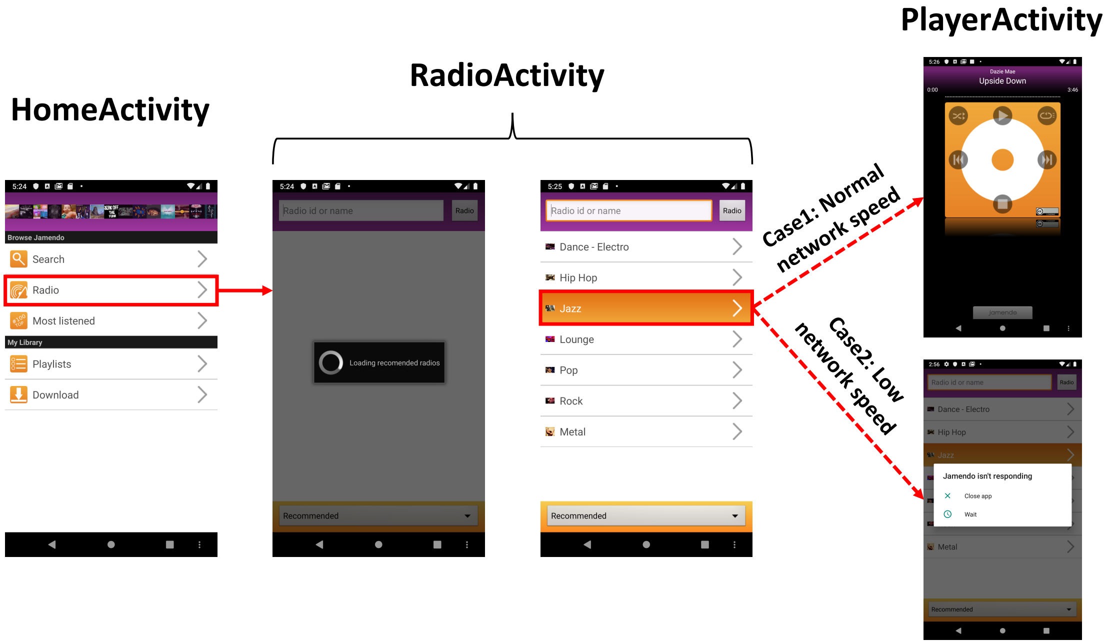


Figure 4.3: Motivation: *Jamendo* music player app.

of our technique, so we devised ways to avoid computing some slices without significant impact on precision (Section 4.1.5).

4.1.1 Motivating Example

In this section, we give a motivating example to illustrate the problem of missing progress indicators, both in terms of visible app behavior and at the source code level. We use an example adapted from the open-source *Jamendo* music player app.[1] Figure 4.3 shows relevant screens from the app, and Figure 4.4 shows the corresponding code.

We first describe a user interaction with a proper progress indicator. In the app's home screen (`HomeActivity` in Figure 4.3), when the user taps on ”Radio”, the

[1] `https://f-droid.org/en/packages/com.teleca.jamendo/`

```java
public class HomeActivity extends
    Activity {
  @Override
  protected void onResume() {
    RadioActivity.launch(HomeActivity.
    this);

  }
}
```

```java
public class RadioActivity extends
    Activity {
  @Override
  public void onCreate(Bundle ...) {
    RadioLoading mRadioLoading = new
    RadioLoading(...);
    mRadioLoading.execute();
    ...
    mRadioListView.
    setOnItemClickListener(
    mRadioLstListener);

  }
  private OnItemClickListener
    mRadioLstListener =
    new OnItemClickListener() {
      playlist = new JamendoNetworkAPI()
    .getPlaylist(...);

      PlayerActivity.launch(
    RadioActivity.this, playlist);

    }
}
```

```java
public class RadioLoading extends
    AsyncTask<...> {

  @Override public void onPreExecute() {
    progressDialog.show();...
  }
  @Override protected void
    onProgressUpdate(...){ ... }
  @Override public Result doInBackground
    (...) {
    recommendedRadios =
      new JamendoNetworkAPI().getRadios
    (...);

  }
  @Override public void onPostExecute(
    Result result) {
    progressDialog.dismiss();...
  }
}
```

```java
public class JamendoNetworkAPI {
  public Playlist getRadios(...){ doGet
    (...) }
  public Playlist getPlaylist(...){
    doGet(...) }
  public String doGet(String url){
    HttpGet httpGet = new HttpGet(new
    URI(url));
    HttpResponse httpResponse =
      new DefaultHttpClient().execute(
    httpGet);
    return httpResponse.getEntity().
    getContent();
  }
}
```

Figure 4.4: Motivation: *Jamendo* music player source code.

RadioActivity is launched (line 4 in Figure 4.4), causing a UI transition. Upon initialization, the RadioActivity starts a RadioLoading background task (line 5 in RadioActivity class). The RadioLoading class extends and overrides the methods of Android AsyncTask class, used for executing long-running tasks on a background thread. The Android Framework first invokes the object's onPreExecute() method on the main thread, which displays a progress dialog (line 3 in RadioLoading class), shown as the first RadioActivity screen in Figure 4.3.

Meanwhile, the framework runs the AsyncTask's doInBackground() method in a background thread. In this case, the code retrieves a list of radio stations by calling into the JamendoNetworkAPI class (line 8), which gets the data via an HTTP GET request (lines 4–9 in JamendoNetworkAPI class). Once the network result becomes available, the progress dialog is dismissed (line 11 in RadioLoading class), and the result, a list of recommended radio stations, is displayed (the second RadioActivity screen in Figure 4.3). The progress indicator shows the user that work is ongoing as the app awaits the result.

In the next user interaction, a progress indicator is not properly employed. A user can tap on one of the radio stations shown by the previous step to play the music in a player UI. The code handles such taps via an OnItemClickListener, attached at line 7 in RadioActivity class of Figure 4.4. The listener directly uses JamendoNetworkAPI to retrieve a playlist (line 11), and then uses the playlist when launching the player (line 12). On a fast network connection, this interaction could proceed smoothly, as shown in Case 1 in Figure 4.3. However, on a slow network connection, the UI will freeze until the playlist is received, with no indication that work is being done. If the request takes too long, the user will see an Application Not Responding (ANR) dialog, as shown in Case 2 of Figure 4.3. This example illustrates the challenge of ensuring progress indicators are always employed when needed; even when a developer is aware of the need for such indicators, it is easy to forget them in some cases.

The first interaction that correctly used a progress indicator (showing the station list) is useful for showing how the work of Kang et al. [58] can miss real bugs. That technique finds issues by looking for cases with *no UI updates at all* for a certain time period. Consider a case where in Figure 4.4, lines 3 and 11 in `RadioLoading` class were removed, so there is no longer a progress indicator shown while downloading the station list. Even with these lines removed, after the user taps the ”Radio” option on the home screen, there would still be an immediate UI update to transition to the `RadioActivity` (line 4 in `HomeActivity` class). The app would then be “stuck” at the blank `RadioActivity` screen until the network response was received. A technique looking for any type of UI update would miss the bug in this case, due to the immediate transition to a new `Activity`. We found this scenario to occur frequently in our evaluation (for 84% of discovered bugs; see Section 4.1.6), justifying our approach of distinguishing progress indicator operations from other UI updates.

4.1.2 Problem Definition and Terminology

In this section, we give a semantic definition of correct progress indicator usage in a program, based on standard notions of program dependencies. We introduce some terminology and background. Then, we present our definition in Section 4.1.3, and discuss some of its practical tradeoffs.

We first introduce some terminology used to define progress indicator correctness.

Our definition refers to the following key operations in programs:

User event handler. A user event handler is the code that executes in response to some user interaction with the application, like a tap or swipe in a mobile app. When referring to a user event handler as a program point, we mean the *start* of the handler (e.g., the entrypoint of a handler callback).

Long-running operation. A long-running operation is any operation that may run long enough to cause a perceptible "lag" in the application, necessitating use of a progress indicator. We do not study the length of time corresponding to perceptible lag here, as this has been studied extensively in previous work, e.g., Kang et al. [58].

Progress indicator. Progress indicator operations control the display of a progress indicator in the user interface, like the spinner in first `RadioActivity` screen in Figure 4.3.

UI update. UI update operations are those that update the visible user interface, *excluding* progress indicator operations.

We assume an *API mapping* is provided that describes which program API calls or statements correspond to the above key operations. For each long-running operation, we assume that the API mapping provides a *start point* and *end point* for the operation. E.g., a start and end point could respectively be the call and return for a long-running synchronous function, or a starting function call could be matched with an ending callback invocation for an asynchronous API. Similarly, we assume a start and end point is provided for each progress indicator, typically API calls to show and hide the indicator.

We also make use of some standard terminology from the program dependence and slicing literature. A *program dependence graph* (PDG) [38] represents dependencies between statements in a program. Dependencies include *data dependencies*, where one statement uses a value computed by another, and *control dependencies*, where one statement determines whether another statement may or may not execute. For program constructs like procedure calls and asynchronous callbacks, more sophisticated dependence graphs have been devised, such as system dependence graphs (SDGs) [50] and hierarchical PDGs [24]. For our purposes, we assume that an appropriate dependence graph has been constructed for the program at hand. We also assume appropriate notions of *reachability* and *paths* exists for the dependence graph, ignoring details of which paths are valid (e.g., paths must have matching calls and returns for SDGs [50]).

Finally, we introduce a notion of *immediate dependence* for our relevant operations. Statement b is *dependent* on statement a if there exists a path from a to b in the dependence graph.[2] Statement b is *immediately dependent* on statement a if there exists a path p from a to b in the dependence graph, such that p contains no statements corresponding to the start of a user event handler or a long-running operation start or end point. We use immediate dependence to avoid conflating dependent user interactions, to be discussed further in Section 4.1.3.

4.1.3 Correct Progress Indicator Usage

We now define correct progress indicator usage for a program P. We first define which sequences of operations require use of a progress indicator. A *relevant interaction* is a tuple $\langle e, l_s, l_n, u \rangle$ with the following properties:

[2] For this discussion, we assume dependence graph edges are in the forward direction.

- e is a user event handler in P;

- l_s is a start point for a long-running operation that is immediately dependent on e;

- l_n is the end point corresponding to l_s; and

- u is a UI update operation immediately dependent on l_n.

A relevant interaction requires a progress indicator since the UI update u is performed due to the completion of the long-running operation (l_s, l_n), which was started due to user event e. Hence, u usually corresponds to the result the user expected by triggering e (further discussion below).

The relevant interaction definition uses immediate dependence instead of standard dependence to avoid conflation of sequenced user events and UI updates. For example, in Figure 4.3, the user first taps to transition from the `HomeActivity` to the `RadioActivity`, and then taps again to get to the `PlayerActivity`. The long-running network request and UI update for the second tap are both dependent on the *first* tap, as neither are possible until the app has transitioned to show the `RadioActivity`. However, we should *not* require a single progress indicator to cover both the UI transitions. The use of immediate dependence avoids this issue, as the second network request and UI update are not immediately dependent on the first `HomeActivity` tap.

A program P makes *correct* use of progress indicators iff for all relevant interactions $\langle e, l_s, l_n, u \rangle$ in P, the following two conditions hold:

1. A progress indicator start point p_s is immediately dependent on e; and

2. The matching progress indicator end point p_n is immediately dependent on l_n.

In other words, the progress indicator should be displayed due to the user event, and only hidden once the long-running operation is complete. We again make use of immediate dependence to avoid conflating sequenced interactions. Figure 4.1 shows the required immediate dependencies graphically.

Discussion Our definition of correct progress indicator usage is designed to make bug finding practical, but it does not perfectly capture all possible progress indicator behaviors. Some correct behaviors are not captured by the definition, potentially leading to false positives in a bug-finding tool. For example, consider an interaction where after a user tap, the application quickly shows the relevant result, but it also performs an asynchronous request for some optional information. (E.g., in a game, a score could be updated immediately, while a server could be asynchronously queried to check if a high score has been exceeded.) In such cases, our definition necessitates use of a progress indicator for the network request, though it is unnecessary.

Our definition ignores time delays in showing progress indicators, possibly leading to missed bugs. Consider a contrived case where in an event handler e, a program starts a long-running operation immediately, but only shows a progress indicator after a five-second delay. This delay is too long, and the app may appear to be unresponsive. However, this code may still meet our definition of correct progress indicator usage, as it only requires p_s to be immediately dependent on e, without considering timing delays. Correctly incorporating delays into the definition is non-trivial—some applications use delays to avoid showing a progress indicator when a network request completes very quickly, and such patterns should not be prohibited.

The definition also assumes that a UI update corresponds to at most one long-running operation. In general this assumption may not hold, as a UI update may only occur after multiple long-running operations, run either sequentially or in parallel. Such cases could lead to false positives or false negatives, but we have not observed them in practice and expect them to be quite rare. If such cases arise more frequently, they could be handled by wrapping the multiple long-running operations inside a higher-level abstraction, and modeling the start and stop APIs of the new abstraction as long-running.

We plan to study improvements to our correctness definition in future work. But, our experimental evaluation showed that our current definition is sufficient for finding a variety of bugs (see Section 4.1.6).

4.1.4 Dynamic Bug Finding Algorithm

In this section, we present an algorithm for finding violations of correct progress indicator usage (as defined in Section 4.1.3) via dynamic slicing. Dynamic slicing tools do not directly support our notion of immediate dependence. The key insight of the algorithm is that in the dynamic setting, we can combine trace ordering with slicing to handle immediate dependence requirements. Our tool ProgressDroid, to be discussed in Section 4.1.5, uses this algorithm (with minor modifications) to analyze Android applications.

Our algorithm FindProgressBugs is shown in Algorithm 2. FindProgressBugs operates on a trace t of some dynamic execution of a program. We assume the trace is indexed, so that the relative ordering of events in the trace can easily be computed. SubTrace(t, s, s') returns the portion of trace t occurring between events s and s', and

Algorithm 2 Algorithm for finding progress indicator bugs using dynamic slicing.

```
 1: procedure FindProgressBugs($t$)
 2:     $U \leftarrow$ UIUpdates($t$)
 3:     for all $u \in U$ do
 4:         $S_u \leftarrow$ BackSlice($u, t$)
 5:         $E \leftarrow$ EventHandlers($S_u$)
 6:         if $E = \emptyset$ then                         ▷ no user interaction involved
 7:             continue
 8:         end if
 9:         $e \leftarrow$ Latest($E$)                        ▷ for immediate dependence
10:         $r \leftarrow$ SubTrace($t, e, u$)
11:         $L \leftarrow$ LongRunningOps($r$)
12:         if $\exists\ (l_s, l_n) \in L \mid e \in$ BackSlice($l_s, t$) $\wedge\ l_n \in S_u$ then
13:             $P \leftarrow$ ProgressOps($r$)
14:             if $\nexists\ (p_s, p_n) \in P \mid (e \in$ BackSlice($p_s, t$) $\wedge$
                                    $l_n \in$ BackSlice($p_n, t$)$)$ then
15:                 ReportBug($e, l_s, l_n, u$)
16:             end if
17:         end if
18:     end for
19: end procedure
```

Latest(S) returns the latest event in S. We assume a routine BackSlice(s, t) for computing the backward (dynamic) slice of an event s from trace t, with the slice represented as a set of trace events. We base the algorithm on backward slicing as that is the primitive provided by the slicing tool AndroidSlicer used by ProgressDroid [24]. Finally, we assume routines EventHandlers and UIUpdates for finding user event handlers and UI updates in a trace, and LongRunningOps and ProgressOps for respectively finding start-stop pairs for long-running and progress operations.

Given these primitives, FindProgressBugs proceeds as follows. First, all UI updates in the trace are found (line 2). For each discovered UI update u, a backward slice S_u is computed (line 4). Lines 5–8 ensure there is some user event handler in S_u; otherwise, the UI update was unrelated to a user interaction. Line 9 selects e as the *latest* user event handler from S_u. This choice ensures that subsequent steps will only consider long-running and progress operations respecting immediate dependence conditions, thereby avoiding conflation with previous interactions.[3]

The remainder of the algorithm fairly closely tracks the definition of correct progress indicator usage in Section 4.1.3. Lines 11–12 search for an appropriate long-running start and stop operation in the sub-trace between e and u. If such an operation is discovered, lines 13–15 search for a corresponding progress indicator and report a bug if it is missing. Again, since our search is limited to the portion of the trace between e and u, we can simply leverage backward slices to find appropriate dependencies, without directly computing immediate dependence.

4.1.5 Bug Finding for Android

Here, we describe how we implemented Algorithm 2 in a tool ProgressDroid for finding progress indicator bugs in Android apps. The structure of ProgressDroid was previously shown in Figure 4.2. Here we detail ProgressDroid's API mappings (Section 4.1.5), extensions to AndroidSlicer [24] for collecting traces and slicing (Section 4.1.5), practical modifications of Algorithm 2 (Section 4.1.5), and input generation strategies (Section 4.1.5). All code for ProgressDroid will be made available as open source upon publication.

[3] This relies on the assumption of one long-running operation per UI update, discussed in Section 4.1.3.

API mappings

As shown in Figure 4.2, ProgressDroid leverages an API mapping to identify which API calls correspond to the key operations from Section 4.1.2. Our API mappings are mostly derived from a thorough study of the APIs made available by the Android Framework. For user event handlers, we re-use the API modeling implemented in AndroidSlicer [24]. For UI updates, our mapping includes over 4,000 built-in UI API methods from the Android framework. Apps often implement their own UI components by sub-classing standard Android framework classes. For each app to be analyzed, ProgressDroid performs a lightweight analysis to find such classes, and includes their overriding APIs in the mapping. Like AndroidSlicer, this analysis is implemented using the Soot framework [105].

For long-running operations, ProgressDroid's current focus is on network requests, the most common type of long-running operation in Android applications. ProgressDroid includes standard APIs for performing network requests in its mapping, including the standard HTTP APIs provided by Android and APIs for fetching URL data for `WebViews`. However, Android apps also make use of a variety of third-party libraries for networking; we have not exhaustively modeled these APIs in ProgressDroid. So, in some cases, ProgressDroid may not automatically identify that a user interaction leads to a network request. Such gaps appeared rarely in our evaluation (see Section 4.1.6). In a commercial setting, a company could easily add mappings for their chosen networking APIs to ProgressDroid to ensure good coverage of their apps. Similar issues exist for progress APIs, and as with network requests, ProgressDroid's mapping contains the most commonly-used APIs that we identified.

Extending AndroidSlicer

ProgressDroid uses AndroidSlicer [24] both to instrument apps to collect dynamic traces and to compute slices over those traces, as shown in Figure 4.2. ProgressDroid relies heavily on AndroidSlicer's ability to slice across asynchronous operations in Android applications, as the relevant dependencies for progress indicator bugs often cut across asynchronous operations.

We found that we had to extend the dependencies modeled by AndroidSlicer in two key ways to make ProgressDroid effective. First, we had to add modeling of the dependencies arising from use of Android's `AsyncTask` type, as it is used often for network requests (e.g., see Figure 4.4). AndroidSlicer only instruments application code, so it relies on models of framework code like `AsyncTask`s to operate correctly. Dependencies introduced via these framework models are termed *semantic dependencies* [24].

The lifecycle of an `AsyncTask` execution is comprised of four steps as shown on the left side of Figure 4.5. First, `onPreExecute()` is called by the framework on the main thread. Second, `doInBackground()` is invoked on a background thread, immediately after `onPreExecute()`, to execute the long-running operation. While running, `doInBackground()` can invoke `publishProgress()` to provide a progress update; this update is rendered in the UI via a call from the framework to `onProgressUpdate()`. Finally, when the long-running task is complete, `onPostExecute()` is invoked to render the result.

The semantic dependencies introduced for `AsyncTask` execution are shown on the right of Figure 4.5. We define a semantic control dependence from `doInBackground()` to onPreExecute() to capture their sequencing.[4] We also define a semantic data dependence from doInBackground() to execute(), for the URLs that execute() passes into the task. Finally, we introduce semantic data dependence edges from onProgressUpdate() to any calls to publishProgress() within doInBackground(), and from onPostExecute() to the returned result of doInBackground(). Note that these semantic dependencies are a template: they are instantiated for every occurrence of these AsyncTask operations in the dynamic trace.

Our second extension of AndroidSlicer was to handle control dependences for Java's try-catch blocks. We found that exception handling code appeared frequently alongside networking and progress indicator code (e.g., to handle network timeouts), necessitating better handling. AndroidSlicer did not already include these dependencies since Soot did not provide exceptional control dependencies in its control-dependence graph. Static exclusion of such dependencies is a common design decision, as nearly every statement in Java may throw an exception, leading to an overwhelming number of dependencies [94]. For our scenario, we avoid this dependence explosion by specializing our handling to observed traces. Given a trace and knowledge of which statements C start catch blocks (computed from the app bytecode), we introduce a control dependence from any statement execution in C to the last statement observed to execute in the corresponding try block.

[4] This semantic control dependence does *not* indicate that onPreExecute() controls doInBackground()'s execution. Instead, it ensures that doInBackground() is control-dependent on any node that onPreExecute() is control dependent on, to capture these dependencies across threads [24].

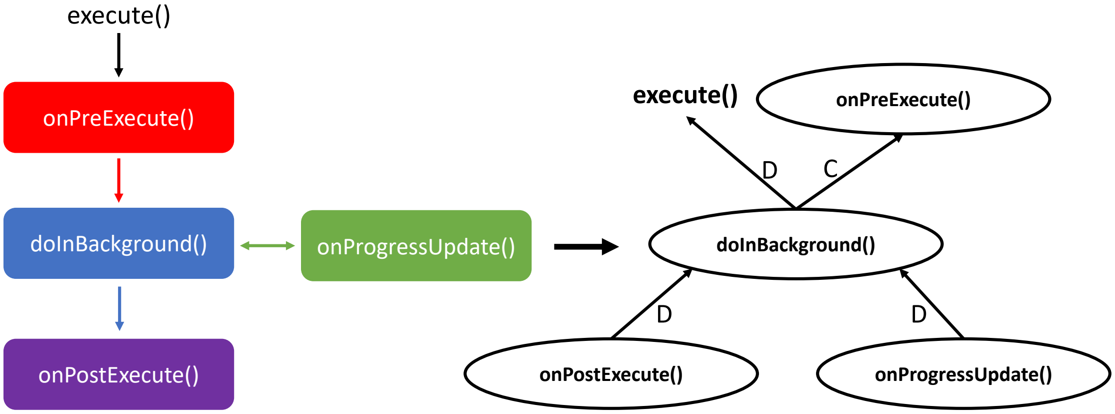


Figure 4.5: Semantic dependencies for *AsyncTask*.

Algorithm Modifications

ProgressDroid uses a modified version of Algorithm 2 when detecting progress indicator bugs in a trace, both for performance and to deal with limitations of our semantic dependence modeling. In general, as apps and traces become larger, computing slices with AndroidSlicer becomes increasingly expensive. So, ProgressDroid skips two of the BackSlice computations shown in Algorithm 2. For line 12, ProgressDroid does not check that the user event handler is in the backward slice of the start of the long-running operation. In practice, if $l_n \in S_u$ and l_s and l_n are matched, we have never observed a case where l_s was not dependent on e.[5]

For Algorithm 2 line 14, ProgressDroid does not perform the slice from the end progress operation p_n to check for the presence of l_n. Similar to the previous case, in our experience, if the progress indicator start operation was dependent on the user event handler, the matched end operation was also dependent on the network request. Another

[5] In principle this could occur, e.g., if the UI update was dependent on some network operation unrelated to the user interaction.

issue for this case is that p_n would typically be control-dependent on l_n, and modeling all the relevant semantic control dependencies in AndroidSlicer for the variety of networking libraries used would be a significant amount of work. In contrast, usually the UI update is data-dependent on l_n (to display the result), making it easier to capture the dependence.

Input Generation

ProgressDroid uses the DroidBot tool [64] to automatically generate inputs to exercise subject apps for trace collection. As shown in Figure 4.2, DroidBot can optionally take as input a manually-written test script for exercising parts of the app. Such scripts were useful for some of our test subjects, e.g., to get past an initial login screen. In an industrial setting, we would expect usage of test scripts to play a large role alongside test generation, as industrial apps often have a scripted test suite for UI interactions.

We added a randomized exploration strategy to complement DroidBot's depth-first and breadth-first strategies [64]. For our goal of finding UI interactions involving long-running network requests, we found disadvantages to both the DFS and BFS approach. DFS would sometimes get stuck deep within an app on a screen unrelated to network interactions, whereas BFS could get stuck attempting to tap every element in a long list view (all of which exercised roughly similar behavior). In our experience, a randomized exploration strategy was more successful in avoiding these states and exploring more network-related behavior. A deeper study of the tradeoffs of these approaches is beyond the scope of this work. We note that even with the randomized approach, we retained DroidBot's ability to replay an interaction sequence of interest, a useful feature for understanding discovered bugs.

4.1.6 Evaluation

We performed an experimental evaluation of ProgressDroid on 30 Android applications, aiming to answer the following research questions:

RQ1. *Effectiveness / Precision*: Can ProgressDroid identify missing progress indicator bugs in real-world Android apps, with a low false-positive rate?

RQ2. *Comparison*: Does ProgressDroid find bugs that would be missed by the previous state-of-the-art [58]?

RQ3. *Scalability*: Does ProgressDroid run in a reasonable amount of time on real-world applications?

Benchmarks. We ran ProgressDroid on 30 apps randomly selected from the top 100 apps in a wide range of categories in the Google Play Store (shopping, food & drinks, lifestyle, etc.). We rejected apps that could not be instrumented by , either due to limitations of or Soot, or due to checksum calculations in the app that prevented modifications. We also rejected apps that did not involve network communication, as ProgressDroid is focused on finding issues involving network requests. Otherwise, the apps were chosen randomly, before any input generation or further exploration of app behavior was performed.

Table 4.1 gives details of our 30 subject apps. The first column shows the app's name. The second column and third column show the app's category and popularity (number of installs, in millions, per Google Play as of August 2019) respectively. The final column gives the app's bytecode size. All the apps have more than 100,000 installations, and the apps are of a significant size, ranging from 1.5MB to 11.5MB (median size = 4.9 MB).

App	**Category**	**Installs** M+	**Dex code size** (MB)
Allama Iqbal Shayari	Books & Ref.	1	4.2
Audiobooks.com	Books & Ref.	1	10.6
Best Love Messages	Lifestyle	0.1	3.5
Bible Hub	Books & Ref.	0.1	1.4
CA DMV	Tools	1	6.2
Cabela's	Business	0.5	10.2
CarWale	Auto & Vehicles	5	11
Classified Listings	Lifestyle	1	3.7
Daily Tamil News	News & Mag.	1	4
Dominos Pizza Carib.	Food & Drinks	0.1	2.9
DOS Careers	Travel & Local	0.1	11.5
Enfermagem	Education	0.5	3.3
Fast Food Specials	Food & Drinks	0.1	8.6
Free Oldies Radio	Music & Audio	1	7
GoodRx Gold	Medical	0.1	1.5
House Plans at Family	House & Home	0.1	2
JackThreads	Shopping	1	2.8
Karmakshetra Online	Education	0.5	5.2
KBB.com	Lifestyle	1	4
Learn English Words	Education	5	7.4
Meeting Guide	Lifestyle	0.1	10.3
NA Meeting Search	Health & Fitness	0.1	8.5
Restaurant.com	Lifestyle	1	3.4
Salud Universal	News & Mag.	0.5	4.7
Startpage Private Search	Lifestyle	0.5	4.3
Stock Quote	Finance	0.5	5
Toyota Financial	Finance	0.5	10.1
VOA News	News & Mag.	1	10
Web Browser & Exp.	Communication	5	7.2
Yogurtland	Food & Drinks	0.1	4.3

Table 4.1: Benchmark: 30 apps.

Environment. All experiments were conducted on the Android emulator running Android version 9, running on an Intel Core i7-4770 CPU 3.4 GHz, 24 GB RAM desktop machine running 64-bit Ubuntu 16.04 LTS. For all the experiments, to drive the app execution, we used DroidBot to send the app 30 UI events. For some of the apps, we used additional test scripts to bypass specific screens/states (e.g., welcome pages, login pages, and text fields requiring specific inputs) to help DroidBot explore deeper app behavior. Such apps are marked with a '*' in Table 4.2.

RQ1. Effectiveness / Precision

Table 4.2 gives key data from our evaluation of ProgressDroid. The most relevant columns for RQ1 are:

- **Meth. cov.:** the percentage of app methods covered by the generated test input;
- **UI slice:** the number of UI update operations from which we perform slices;
- **UI → Net:** the number of cases where the slice showed that the UI update was dependent on a long-running network request; and
- **Error report:** the number of missing progress indicator errors reported by the tool.

Our median method coverage was 5.3%, but we were still able to trigger a significant number of UI update calls per app. In 8 apps (denoted by a '*' in Table 4.2), we wrote manual test scripts to bypass specific screens/states. Still, with better method coverage our tool may be able to discover even more issues. Automated test generation for Android is an

App	**Trace**	**Meth.**	**UI**	**UI**	**Error**	**Static**
	(MB)	**cov.**	**slice**	→	**report**	**UI**
		(%)		**Net**		
Allama Iqbal.	36.7	3.4	23	5	5	1
Audiobooks.com	32.3	7.2	8	3	0	-
Best Love.	149	6.8	22	3	0	-
Bible Hub	4	7.9	20	6	6 (FP)	-
CA DMV (*)	23.2	10.6	47	13	0	-
Cabela's	38.0	31.3	50	2	1	1
CarWale (*)	83.9	5.6	36	11	3	3
Classified. (*)	17.9	7.0	9	6	6	0 (*ad*)
Daily Tamil News	19.9	1.6	7	0	0	-
Dominos Pizza.	4.3	2.6	25	5	4	4
DOS Careers	27.6	4.4	14	4	1	1
Enfermagem	18.5	4	30	8	6	0 (*ad*)
Fast Food Specials	36.4	5.1	62	15	0	-
Free Oldies Radio	16.1	2.5	18	3	0	-
GoodRx Gold (*)	1.5	0.15	9	3	3	1
House Plans.	21.1	2.8	12	3	1	1
JackThreads	9	8.0	7	2	2	0
Karmakshetra.	16.9	1.3	16	5	2	2
KBB.com	53.8	10.9	61	11	0	-
Learn English.	54.5	9.1	9	3	0	-
Meeting Guide	278.7	6.7	18	4	0	-
NA Meeting.	39.4	4.2	8	2	0	-
Restaurant.com (*)	6.7	2.0	14	10	0	-
Salud Universal	4.9	2.7	10	4	4	4
Startpage Private.	8.4	1.1	6	1	0	-
Stock Quote (*)	40.1	1.4	4	3	1 (FP)	-
Toyota. (*)	40.1	7.4	33	4	4	4
VOA News	104	7.1	55	12	0	-
Web Browser. (*)	23.4	7.2	34	6	2	2
Yogurtland	77.3	6.4	34	6	5	5
min	1.5	0.15	4	0	0	0
median	25.5	5.3	18	4	1	2
max	278.7	31.3	62	15	6	6

Table 4.2: ProgressDroid evaluation: core results.

orthogonal problem to ours and an active area of research [72, 44]; ProgressDroid can easily integrate improved techniques as they are developed.

ProgressDroid reported a total of 56 missing progress indicator bugs, with at least one error report on 17 of the 30 applications. We manually inspected all of the reported bugs, leveraging DroidBot's ability to reproduce the same event sequence used during testing [64]. We also artificially reduced the bandwidth of the test emulator, to increase the time required for network requests and make bugs more evident. We confirmed all except seven reported progress indicator bugs to be a true positive error. The exceptions were from the *Bible Hub* and *Stock Quote* apps, which indicate progress to the user via a `TextView` with a ”Loading...” text label, a scenario that was not included in our API model. Our API model could easily be extended to handle this scenario.

Also, in one app (*Daily Tamil News*), we observed that ProgressDroid did not correctly identify the network request calls, due to a bug in the instrumentation phase. In this case it did not lead to false negatives, as *Daily Tamil News* uses progress indicators correctly, but it could have caused false negatives in a buggy app.

In general, a wide variety of progress indicator and networking APIs are used in Android applications, including custom APIs for individual apps. In testing more apps with our current API models, we expect ProgressDroid would have occasional false positives due to un-modeled progress APIs and false negatives due to un-modeled network APIs. We believe that having a pluggable and extensible API model is critical for real-world usage scenarios, so developers can easily handle these new APIs as they arise.

RQ2. Comparison with Previous Work

We compared ProgressDroid against the approach of the *Pretect* tool of Kang et al. [58], the only previous work we are aware of on detecting unresponsive Android UIs. *Pretect* works by monitoring app execution at a system level to detect cases where after some user input, there is a *complete* absence of UI updates for some pre-defined period of time. This approach can suffer from false negatives when some non-progress-indicator UI update occurs after the user input, as described in Section 4.1.1. Unfortunately, we were unable to obtain the *Pretect* tool or source code.[6] So, to compare with *Pretect*, we investigated all true positive issues reported by ProgressDroid, counting cases where some unrelated UI update would have caused *Pretect* to miss the issue.

The relevant results for this comparison appear in Table 4.2, in the "Static UI" column. The column reports the number of cases with missing progress indicators for which there is an immediate screen transition after the user event (like to a new activity), but then no progress indicator appears. This is the primary type of unrelated UI update we observed that would affect *Pretect*. Additionally, some apps include ads, which can animate independently of the user interaction, leading to unrelated UI updates. We note two cases in the table where the app does not have a static UI update but does show ads during the interaction with a missing indicator.

Overall, for all apps with an observed bug except for one, at least one of the missing indicator issues involved a static UI update or an advertisement. As a result, 41 out of the 49 bugs that ProgressDroid found (84%) would not have been found by *Pretect*. These

[6] We were unable to access the URL given for the tool in Kang et al. [58].

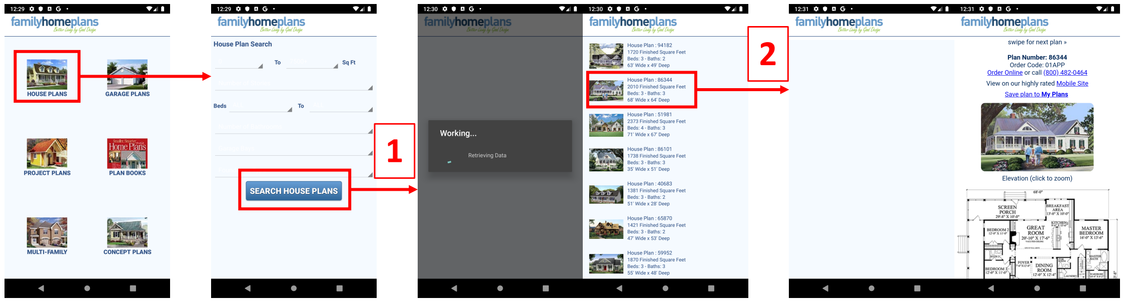
Network API call	**Number of bugs**
AsyncTask helper class	34
Volley HTTP library	10
WebView web content displayer	12

Table 4.3: Network API categorization of apps with missing progress indicator detected by ProgressDroid.

data indicate that tracking all UI updates without distinguishing progress indicator APIs leads *Pretect* to miss a significant number of progress indicator bugs.

Additionally, ProgressDroid can leverage its API mappings to provide more details on which operations were involved when a progress indicator is missing. Table 4.3 shows a categorization of discovered progress indicator bugs based on which API was used to manage the background network request. This kind of information could speed bug fixing—for example, for `AsyncTask`, typically a proper implementation of `onProgressUpdate()` can be used to fix the problem. In contrast, *Pretect* can flag which user interaction led to the UI update delay, but cannot automatically provide details on what long-running operation caused the delay.

Example. We discuss one issue in detail from the *House Plans at Family Home* app, to make more concrete the workings of ProgressDroid and the distinction with *Pretect*. Figure 4.6 shows screens for two user interactions and corresponding UI transitions for the app (numbered 1 and 2), with excerpts from the corresponding dynamic traces appearing below. In transition 1, when the user taps on the “SEARCH HOUSE PLANS” button, a progress indicator is shown after the activity transition, while the network operation is running. Upon the completion of the operation, a list of houses is shown to the user.



```
10521:onClick:com.nvc.familyhomeplans.HousePlanSearchPage$9:$r3 = virtualinvoke $r2.<com.nvc.familyhomeplans.HousePlanSearchPage: android.app.Application getApplication()>()
[...]
10750:loadPlans:com.nvc.familyhomeplans.PlanListPage:$r1 = staticinvoke <android.app.ProgressDialog: android.app.ProgressDialog show
(android.content.Context,java.lang.CharSequence,java.lang.CharSequence,boolean,boolean)>($r0, " Working...", " Retrieving Data", 1, 0)
[...]
66778:getView:com.nvc.familyhomeplans.PlanListAdapter:$r11 = $r5.<com.nvc.familyhomeplans.data.vo.PlanVO: java.lang.String imageURL>
66779:getView:com.nvc.familyhomeplans.PlanListAdapter:virtualinvoke $r13.<com.nvc.familyhomeplans.ImageDownloader: void download(java.lang.String,android.widget.ImageView)>($r11,
$r12)CAUGHT_FIRST_INST
66798:download:com.nvc.familyhomeplans.ImageDownloader:specialinvoke $r0.<com.nvc.familyhomeplans.ImageDownloader: void forceDownload(java.lang.String,android.widget.ImageView)>($r11, $r2)
66820:forceDownload:com.nvc.familyhomeplans.ImageDownloader:virtualinvoke $r7.<com.nvc.familyhomeplans.ImageDownloader$BitmapDownloaderTask:android.os.AsyncTask execute(java.lang.Object[])>($r8)
66822:doInBackground:com.nvc.familyhomeplans.ImageDownloader$BitmapDownloaderTask:$r3 = $r0.<com.nvc.familyhomeplans.ImageDownloader$BitmapDownloaderTask: java.lang.String url>")
66823:doInBackground:com.nvc.familyhomeplans.ImageDownloader$BitmapDownloaderTask:$r1 = virtualinvoke $r4.<com.nvc.familyhomeplans.ImageDownloader: android.graphics.Bitmap
downloadBitmap(java.lang.String)>($r3)")
66854:downloadBitmap:com.nvc.familyhomeplans.ImageDownloader:specialinvoke r29.<org.apache.http.client.methods.HttpGet: void <init>(java.lang.String)>($r13)
66855:downloadBitmap:com.nvc.familyhomeplans.ImageDownloader:r31 = (org.apache.http.client.HttpClient) $r16
66856:downloadBitmap:com.nvc.familyhomeplans.ImageDownloader:$r19 = interfaceinvoke r31.<org.apache.http.client.HttpClient: org.apache.http.HttpResponse
execute(org.apache.http.client.methods.HttpUriRequest)>(r29)
[...]
66868:access$000:com.nvc.familyhomeplans.PlanListPage:r1 = $r0.<com.nvc.familyhomeplans.PlanListPage: android.app.ProgressDialog dismiss()>
[...]
66879:onPostExecute:com.nvc.familyhomeplans.ImageDownloader$BitmapDownloaderTask:$r2 = (android.graphics.Bitmap) $r1
66880:onPostExecute:com.nvc.familyhomeplans.ImageDownloader$BitmapDownloaderTask:virtualinvoke $r0.<com.nvc.familyhomeplans.ImageDownloader$BitmapDownloaderTask: void
onPostExecute(android.graphics.Bitmap)>($r2)
66893:onPostExecute:com.nvc.familyhomeplans.ImageDownloader$BitmapDownloaderTask:$r5 = $r0.<com.nvc.familyhomeplans.ImageDownloader$BitmapDownloaderTask: java.lang.ref.WeakReference
imageViewReference>
66895:onPostExecute:com.nvc.familyhomeplans.ImageDownloader$BitmapDownloaderTask:$r6 = virtualinvoke $r5.<java.lang.ref.WeakReference: java.lang.Object get()>()
66896:onPostExecute:com.nvc.familyhomeplans.ImageDownloader$BitmapDownloaderTask:$r7 = (android.widget.ImageView) $r6
66909:onPostExecute:com.nvc.familyhomeplans.ImageDownloader$BitmapDownloaderTask:virtualinvoke $r7.<android.widget.ImageView: void setImageBitmap(android.graphics.Bitmap)>($r2)
```

User interaction

Progress indicator start point

1

Progress indicator end point

UI update

```
72450:onItemClick:com.nvc.familyhomeplans.PlanListPage$returnClickedItem:$r3 = $r0.<com.nvc.familyhomeplans.PlanListPage$returnClickedItem: com.nvc.familyhomeplans.PlanListPage this$0>
[...]
73463:moveToState:android.arch.lifecycle.LifecycleRegistry:$r0.<android.arch.lifecycle.LifecycleRegistry: boolean mHandlingEvent> = 0
73464:<init>:com.nvc.familyhomeplans.PlanDetailPage:specialinvoke $r0.<android.support.v7.app.AppCompatActivity: void <init>()>()
73479:onCreate:com.nvc.familyhomeplans.PlanDetailPage:specialinvoke $r0.<android.support.v7.app.AppCompatActivity: void onCreate(android.os.Bundle)>($r1)
[...]
73705:onCreate:com.nvc.familyhomeplans.PlanDetailPage:virtualinvoke $r20.<java.lang.StringBuilder: java.lang.StringBuilder
append(java.lang.String)>("http://www.familyhomeplans.com/mobile/mobileDetail.cfm?switch=")
73782:onCreate:com.nvc.familyhomeplans.PlanDetailPage:virtualinvoke $r20.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>($r6)
73783:onCreate:com.nvc.familyhomeplans.PlanDetailPage:virtualinvoke $r20.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>("&token=")
73784:onCreate:com.nvc.familyhomeplans.PlanDetailPage:virtualinvoke $r20.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>($r22)
73785:onCreate:com.nvc.familyhomeplans.PlanDetailPage:$r6 = virtualinvoke $r20.<java.lang.StringBuilder: java.lang.String toString()>()
73786:onCreate:com.nvc.familyhomeplans.PlanDetailPage:$r11 = $r0.<com.nvc.familyhomeplans.PlanDetailPage: android.webkit.WebView webView>
73787:onCreate:com.nvc.familyhomeplans.PlanDetailPage:virtualinvoke $r11.<android.webkit.WebView: void loadUrl(java.lang.String)>($r6)
```

User interaction

Static UI update

2

UI update

Figure 4.6: App screens and trace details for the *House Plans at Family Home* app, including (1) a correct transition and (2) a bug that would be missed by *Pretect*.

In the corresponding trace excerpt, a UI update call to `setImageBitmap` is the slicing criterion. From the trace, we can see that `setImageBitmap` is called from `onPostExecute()` of `ImageDownloader.BitmapDownloaderTask`, which is a subclass of `AsyncTask`. The semantic dependence support for `AsyncTask` described in Section 4.1.5 enables ProgressDroid to reason about dependencies between different `AsyncTask` functions. In this case, our extended support enables to discover that the object reference stored in register $r2 at line number

66909 contains the image bitmap stored variable $r1 at line 66823, the result of a network request. The corresponding user event handler for the UI update is the `onClick` callback at line 10521, and we can see the progress indicator start and stop calls (lines 10750 and 66868 respectively) between the event handler start and UI update trace entries.

For transition 2, when a house is selected by the user, the activity transition occurs without displaying any progress indicator. Here, ProgressDroid creates the slice with respect to the UI update at line 73787 in the second trace excerpt in Figure 4.6. The latest user interaction found in the slice appears at line 72450. Since there are no progress indicator operations in the relevant trace range, ProgressDroid reports a bug. However, *Pretect* would miss this bug since upon the user interaction, an immediate transition happens to the next activity, visible in the trace as the the `onCreate()` call at line 73479, labeled "Static UI update."

RQ3. Scalability

In general, we found that ProgressDroid could scale to moderate length executions (the 30 UI events used in our tests), but longer executions still pose challenges. The running time of ProgressDroid increased proportionately to the trace size, and slicing with was by far the most expensive computation performed by ProgressDroid. Trace sizes for all our tests are shown in Table 4.2. For the app with the largest trace in our tests (*Meeting Guide*, with a 278.7MB trace), instrumentation took 29 seconds, test execution took 150 seconds, and analyzing the trace and computing slices took 49 minutes and 17 seconds.

We took some steps to reduce the cost of the expensive slicing step. First, as described in Section 4.1.5, we elide computing some slices entirely, as we found empirically that they were not worth the cost. Also, we manually filtered out trace events from 21 well-known libraries (for advertising, analytic, reporting, authentication, etc.) that are unrelated to our problem. In future work, we plan to improve scalability by performing multiple short executions with DroidBot, using randomized exploration with different seeds, and then analyzing these traces in parallel.

4.1.7 Related Work

To our best knowledge, Kang et al. [58] were the first to propose an automatic tool for discovering missing progress indicators. We have compared our work to theirs in Section 4.1.1 and Section 4.1.6 of the paper. We believe our approaches are complementary. An advantage of Kang et al.'s approach is that it requires no API mapping, so it can find unresponsiveness independent of the underlying long-running operation. But, this lack of API knowledge can lead their technique to miss many bugs, as shown in Section 4.1.6. Another advantage of the Kang et al. approach is that they need not instrument applications, since they observe app behavior at a system level. Instrumentation of production APKs can fail for a variety of reasons, as discussed in Section 4.1.6, though we expect this to be much less of a problem in real-world developer usage (since the developers have the source code and control the entire build process).

Previous work has studied automatic detection of other types of user interface bugs, such as web page layout failures [106, 70, 15, 88] and accessibility issues [84]. Other work detects potential crashes from performing user interface updates on a background thread [96, 42]. But, other than Kang et al., we are unaware of previous work capable of detecting missing progress indicator bugs.

Our work has interesting relationships to detection of event-based race conditions for web and mobile applications [51, 71, 26, 86]. Like event races, progress indicator bugs involving slow network requests may be difficult to identify with testing in a standard developer environment, as network connections tend to be very fast. Detecting both types of bugs require tools that can precisely model asynchronous application behaviors. More recently, static analysis approaches have been devised for detecting event races [89, 53]. In future work we plan to investigate static approaches to detecting common progress indicator bugs as well.

Android's Strict Mode [3] prevents certain types of I/O operations from running on the main thread. When such operations run on the main thread, they block the user interface from updating, potentially leading to an ANR dialog [4] (see the example in Section 4.1.1). While Strict Mode can help catch some cases of unresponsiveness, it does not give any warning when I/O is performed on a background thread without a corresponding progress indicator.

Lin et al. [65] present a refactoring to transform usages of `AsyncTask` to types like `IntentService` more suitable for long-running tasks. Their refactoring also translates the progress indicator code from an `AsyncTask` into corresponding `IntentService` code. It would be interesting future work to see if such an approach could also detect and introduce progress indicator code when it is missing.

A wide variety of work studies specification mining [19], which can learn API usage patterns from program executions or from static analysis or machine learning over large code bases [92, 78]. Progress indicator bugs often involve typestate of multiple related objects (the progress indicator, the network request, UI data structures), and learning of multi-object specifications is not supported by most techniques (the work of Pradel et al. is an exception [85]). Even so, specification mining could be a useful way to handle the multitude of networking and progress indicator APIs used in practice without needing handwritten models.

4.1.8 Summary

Missing progress indicator bugs are a serious problem in modern applications, but understudied by the software engineering community. We have described a novel technique for discovering missing progress indicator bugs based on program semantics, in particular program dependencies. The technique was implemented in a tool ProgressDroid, which found several bugs in real-world Android applications that would have been missed by previous techniques. Our hope is that our semantic definition of these bugs and our implemented tool will lead to more future work on addressing this type of user interface issue.

4.2 Vulnerabilities in Identifying Unique Devices in Android

In this section, we briefly show a new type of vulnerability in unique device identification techniques used in mobile apps which can lead to financial loss. We leverage dynamic slicing technique to identify the unique device identifier parameters. We show how one can get an illicit financial benefit of creating fake activations from a vulnerable unique device identification mechanism. Lastly, we present the future research direction in this domain.

First we provide some background knowledge of unique device ID generation approaches used in Android apps and then we give a brief overview of the possible vulnerability in mobile apps that use multiple distributors/promoters.

4.2.1 Unique Device Identification in Android

The Android OS offers different types of IDs with different behavior characteristics for various test cases. Based on Android documentation [1], there are four types of IDs with respect to the resettability and persistence feature of the ID:

1) Session-only: A new ID is generated every time the user restarts the app.

2) Install-reset: ID survives app restarts but a new ID is generated every time the user re-installs the app.

3) FDR-reset: ID survives app re-installs but a new ID is generated every time the user factory resets the device.

4) FDR-persistent: ID survives factory resets.

The later one is used when developers need to identify a unique first installation of the app. There are many reasons that developers may need to identify a unique first installation of the app rather than a unique user registered to the app. Identifying the first installation of the app requires gathering device information.

In general, several solutions exist for identifying unique ID in Android apps but none is perfect. Different solutions and their advantages and disadvantages are categorized in Table 4.4. As shown in Table 4.4, none of the approaches are perfect and reliable enough for identifying a unique device in Android. Hence, if developers want to absolutely identify a particular device physically, they usually combine different approaches/parameters to

ID	Approach	Advantages	Disadvantages
Unique Telephony Number (IMEI, MEID, ESN, IMSI)	To retrieve this ID, *TelephonyManager.getDeviceId()* is used; it returns IMEI in the case of GSM phones and MEID or ESN in the case of CDMA phones	- Survives to re-installation, rooted, or factory reset	- Limited to smarthphones (i.e., device should use a SimCard) - For dual sim devices, we get two different values - Needs *android.permission. READ_PHONE_STATE*
MAC and Bluetooth address	Unique identifier for devices with Wi-Fi or Bluetooth hardware	- Survives factory reset	- Not all devices have Wi-Fi connections - Does not work for newer versions of Android (Android 6.0 and up) - *WifiManager.getScanResults()*, *BluetoothDevice.ACTION _FOUND*, and *BluetoothLeScanner.startScan (ScanCallback)* are needed
Serial Number	To retrieve this ID, *android.os.Build.SERIAL* is used	- Works in devices without telephony services	- Not all android devices have a serial number
Secure Android ID	To retrieve this ID, *Settings.Secure.ANDROID_ID* is used	- Available for both smartphones and tablets	- Changes after factory reset - Buggy implementation
GUID	To retrieve this ID, *UUID.randomUUID()* is used	- Good solution for particular installation	- Changes per installation

Table 4.4: Unique ID approaches in Android apps.

generate a customized unique ID.

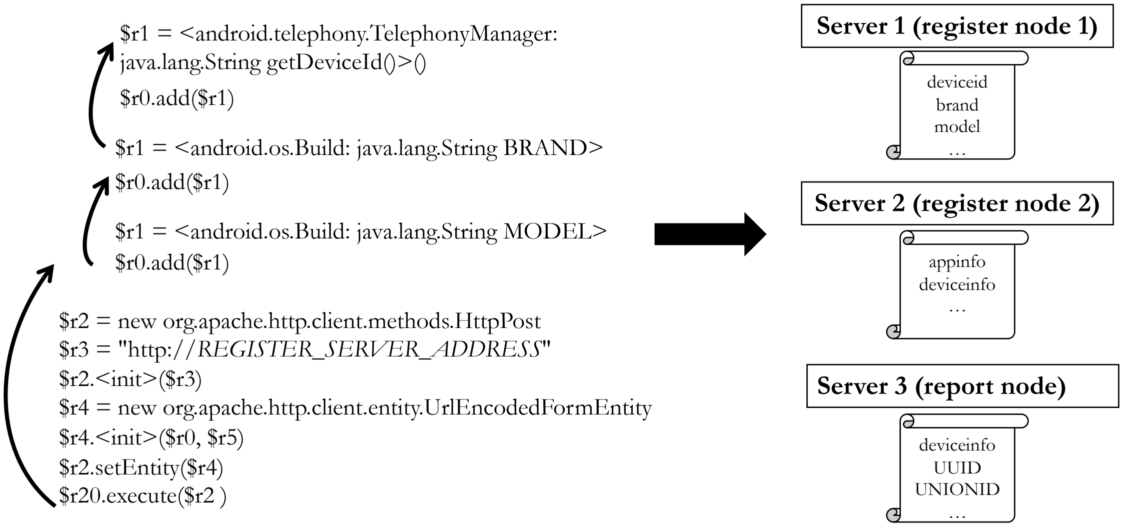
4.2.2 Distribution via Multiple Channels

Mobile apps can be distributed via several channels such as different App stores or third party app promoters. For non-free promoters, the amount of financial transactions between the mobile app developer/vendor and the promoter is based on the number of devices that have installed the app. Hence, the app developers/vendors who want to distribute their app via multiple channels need to have a mechanism to track the number of devices that have installed their app. None of the existing approaches is perfect and reliable for the purpose of identifying unique device installation. Hence, developers implement their own approaches to generate these unique IDs. Next, we show the possible vulnerability in such mechanisms.

4.2.3 Vulnerabilities in Unique Device Identifiers

To show the potential vulnerabilities in unique device identification techniques, we study several Android apps from one of the largest shopping companies in China. We anonymize the name for security concerns. For this purpose, we aim to find the relevant parameters that the app sends to the servers over the first network communications between two. We use AndroidSlicer [24] both to instrument the apps to collect dynamic traces and to compute slices over those traces with respect to the slicing criteria. For slicing criteria, we use the point that the app sends parameters to the server. We illustrate this in Figure 4.7. The slice (left) points out the important instructions in the slice with respect to the network *execute()* call argument register (*$r2*). The register (*$r2*) is data dependent on register (*$r4*) as the URL entity register which is constructed from various parameters (e.g., *deviceid*, *brand*, *model*, etc). We repeat the slicing for the first 3 different network requests that the app makes within the first moments of execution. We collect the slices for all 3 network communications as shown in Figure 4.7 (right). After this, we repeat the requests using fake and still legitimate-look values (for example the same size and format as the original values). Our experiments show that 25 out of 160 (16%) randomly fake activations were successful and were considered as a new installation on a unique new device by the company. Hence, an attacker can cause serious financial losses by repeatedly activating fake devices on the company's servers.

In this section, we presented a new type of potential vulnerability in identifying app installations on unique devices in Android and we showed how we can use a dynamic program analysis technique such as slicing to detect the unique ID generation process and



```
$r1 = <android.telephony.TelephonyManager:
java.lang.String getDeviceId()>()
$r0.add($r1)
$r1 = <android.os.Build: java.lang.String BRAND>
$r0.add($r1)
$r1 = <android.os.Build: java.lang.String MODEL>
$r0.add($r1)

$r2 = new org.apache.http.client.methods.HttpPost
$r3 = "http://REGISTER_SERVER_ADDRESS"
$r2.<init>($r3)
$r4 = new org.apache.http.client.entity.UrlEncodedFormEntity
$r4.<init>($r0, $r5)
$r2.setEntity($r4)
$r20.execute($r2 )
```

Figure 4.7: Analyzing UUID parameters generation via slicing in the vulnerable app.

then re-generate the process using fake values to fool the server into perceiving the fake activation as a normal unique new activation.

Nonetheless, there exist a wide range of potentially interesting avenues for future research in this domain. For example, another use case of unique IDs is for apps with initial discounts (e.g., free delivery for first order, free items for new users of the app, etc). An interesting research direction is to analyze the mechanisms that these apps developers/vendors use to generate the initial discounts for legitimate users via unique device identification mechanisms and identify the potential vulnerabilities in these techniques.

Chapter 5

Efficient Genetic Algorithm for Graph Anonymization

In this chapter, we address the issue of preserving user privacy in graph data anonymization techniques. As it has been shown that state-of-the-art graph anonymization techniques suffer from a lack of strong defense against De-Anonymization (DA) attacks mostly because of the bias towards utility preservation, we propose *GAGA*, an Efficient Genetic Algorithm for Graph Anonymization, that simultaneously delivers high anonymization and utility preservation.

5.1 Background and Motivation

Preserving data privacy has been widely studied. One of the main approaches used to preserve data privacy is based upon the concept of anonymity. Graphs and databases have played an important role in this domain [75, 25, 69, 12, 23]. In this work, we address the data privacy preservation in graphs, specifically for graphs representing social networks. A number of graph anonymization techniques have been proposed to preserve users' privacy. We discuss the limitations of these techniques first from the perspective of defense against DA attacks and then from the perspective of utility preservation to motivate our approach.

As a concrete motivating example, consider the sample graph shown in Figure 5.1. The social graph on the left side is going to be publicly published. Assume that an adversary knows that Alice has 2 friends and each of them has 4 friends, then the vertex representing Alice can be re-identified uniquely in the network (black vertex in Figure 5.1). The reason is that no other vertices have the same *2-neighborhood* structure to the *2-neighborhood* structure for Alice. Existing graph anonymization techniques fail to anonymize this example graph so that an adversary cannot re-identify any user certainly. The *k-degree-anonymity* based algorithm in [67] removes/adds edges from/to the original graph to create a graph in which for every vertex there are at least *k-1* other vertices with the same degree. Based on *k-degree-anonymity*, the graph is *2-degree-anonymized.* In the other approach, *k-neighborhood-anonymity* based algorithm in [120] adds edges to the original graph to create a graph in which for every vertex there are at least *k-1* other vertices with the same *1-neighbourhood* graphs. Based on *k-neighborhood-anonymity*, the graph is *2-neighborhood-anonymized.* Hence, the existing *k-anonymity* approaches are inadequate when the attacker has more complex knowledge about the neighborhood structures.

Based upon the above discussion, we conclude that we must support *k(d)-neighborhood-anonymity* for any *d*, instead of *k-degree-anonymity* or *k-neighborhood-anonymity* for *d=1-neighborhood* considered in prior works. That

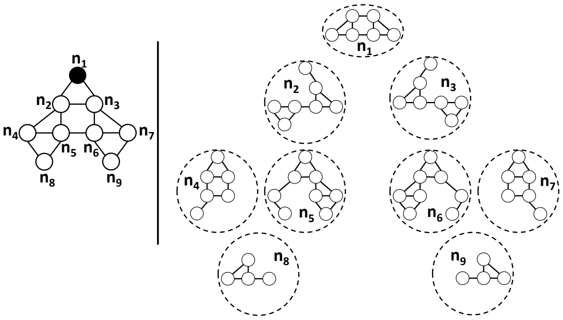


Figure 5.1: Graph to be publicly published on the left and *2-neighborhood* structures for each vertex on the right. Black vertex represents Alice.

is, our approach will provide an algorithm that efficiently enables *d-neighborhood* privacy preservation for any *d* to protect against attacks that use complex neighborhood acknowledgements of the target vertex.

Next we consider the issue of utility preservation. *SecGraph* introduced by Ji et al. [56], evaluates different anonymization algorithms using various utilities. According to their study, *k-neighborhood-anonymity* preserves most of the graph and application utilities. The one application utility which *k-neighborhood-anonymity* algorithm cannot preserve is the *role extraction* utility where it considers the uniqueness of each vertex based on their structure in the graph. Among all anonymization algorithms, the *Rand Switch* approach introduced in [115] where existing pair of edges are switched randomly *n* times, is the only one that can preserve *role extraction.*

Because of the above reason, we give higher priority to edge switching over edge adding and removing since *edge switching* can effectively preserve degree and its related utilities (e.g., role extraction) leading to preserving more utilities. We further apply *edge switching* to the *k(d)-neighborhood-anonymity* model and use Genetic Algorithm as the main approach for utility preservation.

Summary: With more knowledge about the local neighborhood structures in a social network, an adversary has more chances to re-identify some victims. We show that existing anonymization techniques not only do not present a complete model to *defend against DA attacks*, specially structure-based attacks, but also they *fail to make minimum number of changes.* In contrast, an additional goal of our approach is applying fewer changes and thus providing a better trade-off between anonymization and utility preservation.

As a motivating example, Figure 5.2(a) depicts the original graph, and the anonymized graphs generated by our approach, *k-degree Anonymization*, and *Random Walk Anonymization* techniques using the minimum values for the parameters of each approach (*k=2* and *d=2* for our approach, *k=2* for *k-degree Anonymization*, and *r=2* for *Random Walk Anonymization*). Assume an adversary knows that a user has 3 friends and only one of them has another friend, then the user can be re-identified easily (colored bigger vertex in Figure 5.2(a)), since this is the only user with that friendship neighborhood structure.

(Our Approach) In Figure 5.2(b), our approach applies minimum number of changes of 3 edges switches and 1 edge removal to the original graph (i.e., we preserve degrees for all vertices except for only two vertices). We anonymize the graph in a way that for each vertex there is at least one other vertex with similar *2-neighborhood* structure (i.e., there is another user with similar *2-neighborhood* friendship to the target user *2-neighborhood* depicted with two colored vertices which reduces the re-identification chance by 50%).

(*K-degree Anonymization*) In Figure 5.2(c), by applying slightly more changes compared to our approach, the *k-degree-anonymity* concept introduced in [67] is achieved

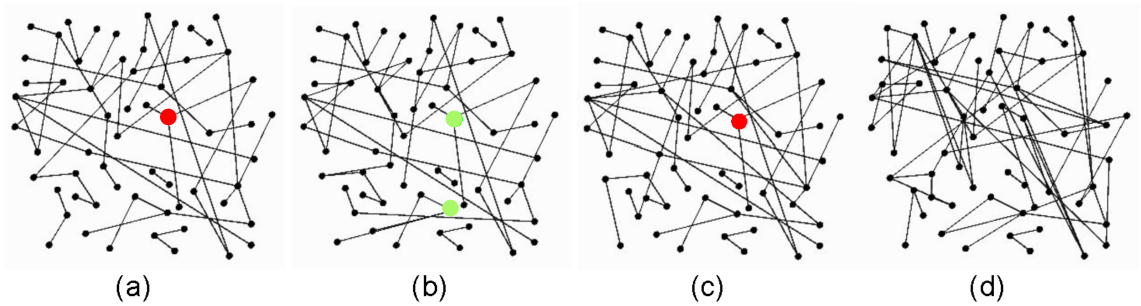


Figure 5.2: Sample social network graph (a) Original graph to be publicly published, bigger colored node shows the victim vertex (b) Anonymized graph using our approach with *k=2* and *d=2*, bigger colored nodes show the victim vertex and the vertex with similar 2-neighborhood (c) Anonymized graph using *k-degree Anonymization* in [67] with *k=2* (d) Anonymized graph using *Random Walk Anonymization* in [76] with *r=2*.

which is weaker in comparison to *k(d)-neighborhood-anonymity*. This means that for each vertex there is at least one other vertex with similar degree which is already satisfied with our approach. Hence, the adversary can still re-identify the target user easily.

(***Random Walk Anonymization***) In Figure 5.2(d), while introducing much more noise compared to our approach, this technique only ensures some level of link privacy. The reason of comparing our approach with *Random Walk Anonymization* technique is that it is the only graph anonymization technique which takes the concept of neighborhoods structures into consideration. That is, in social network graph G, replace an edge *(u,v)* by the edge *(u,z)* where z denotes the terminus point of a random walk algorithm. As a result, noise is introduced into the graph leading to huge data loss.

5.2 GAGA

In this section, we present an Efficient Genetic Algorithm for Graph Anonymization (*GAGA*). *GAGA* creates an optimal anonymized graph by applying minimum number

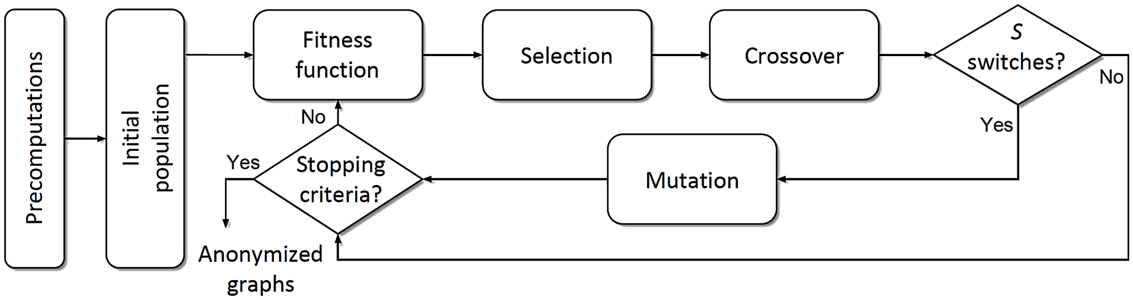


Figure 5.3: *GAGA* overview.

of changes to the original graph in comparison to existing approaches which make the graph less responsive to various queries. *GAGA* can preserve data privacy against many complicated DA attacks. To achieve these goals we use Genetic algorithm (GA) as the main approach. GAs are optimization algorithms that have been applied on a board range of problems like classification, game theory, bioinformatics etc. In this section, we describe how we apply GA to the graph anonymization problem. Figure 5.3 shows an overview of *GAGA*. Now we discuss each step of GA that we used in *GAGA*:

5.2.1 Precomputation Step

Before applying the GA to the original graph, we perform precomputations that evaluate the original graph so that we can choose the best parameters to create the optimal *k(d)-neighborhood-anonymized* graph. As a result, the original graph is categorized as one of: *good*, *bad*, or *ugly* scenarios. In the *good* scenario, the original graph is close to a *k(d)-neighborhood-anonymized* solution and hence it needs a small number of changes. In the *bad* scenario, many vertices do not satisfy the *k(d)-neighborhood-anonymity* and hence the original graph needs changes to large number of vertices. In the *ugly* scenario, few vertices violate the *k(d)-neighborhood-anonymity* but they have a very different neighborhood compared to other vertices; hence it requires huge changes to a small number of vertices. Our precomputations involve the following steps:

Step 1. Percentage of violating vertices: We identify the vertices that violate the *k(d)-neighborhood-anonymity* (i.e., there are less than k-1 other vertices with similar *d-neighborhood* to the *d-neighborhood* of these violating vertices) and compute the percentage of violating vertices. A low percentage of violating vertices means that by applying some changes to a small group of vertices, we can create a *k(d)-neighborhood-anonymized* graph. We further observe that the changes can be small or big themselves. Hence, we consider a threshold value (T_{pv}) and if the percentage is below the threshold value, we consider the graph as one of the *good* ones (i.e., small changes to small number of vertices are required) or an *ugly* one (i.e., big changes to small number of vertices are required). If the percentage is above the threshold value, we consider the graph as *bad* (i.e., some changes to large number of vertices are required).

Step 2. Violating vertices' neighborhoods analysis: After the previous step, the original graph is categorized as *good/ugly* or *bad.* To distinguish between *good* and *ugly* scenarios, we analyze the neighborhoods around violating vertices and compare them with the neighborhoods of vertices that satisfy the *k(d)-neighborhood-anonymity.* If some of the violating vertices have a very different neighborhood than others (we simply compare degrees for this purpose), we categorize the graph as *ugly.* Otherwise, we categorize the graph as *good.* To analyze the rate of difference, we again define a threshold value (T_u) so that if the value is above the threshold we consider the graph as *ugly* scenario. Otherwise, if

R0.60

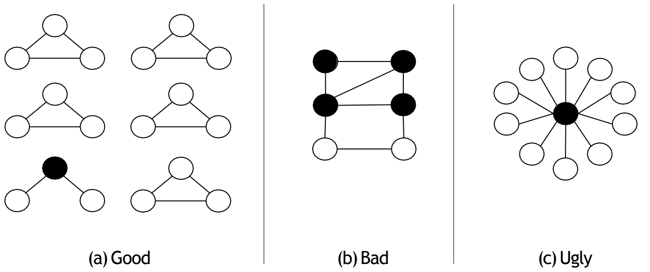


Figure 5.4: Black vertices represent the violating vertices. Assume that k=2, d=1, and T_{pv}=10%. (a) *Good scenario*: 5% of vertices violate the *2(1)-neighborhood-anonymity.* (b) *Bad scenario*: 66% of vertices violate the *2(1)-neighborhood-anonymity.* (c) *Ugly scenario*: 9% of vertices violate the *2(1)-neighborhood-anonymity* but the violating vertex has a very different neighborhood than other vertices.

the value is below the threshold, we consider the graph as *good* scenario. We illustrate the scenarios using three sample graphs in Figure 5.4. The treatment for each of three scenarios is described next:

Good Scenario. In the *good* scenario, in the beginning of the GA process, we focus only on violating vertices according to a probability and apply the GA to them. For this purpose, we select the vertices –selection in GA– from violating vertices to apply the changes (switches, adds, removes) –mutation in GA– so the number of violating vertices will decrease. As we proceed forward towards the end of the process, we select some vertices from other non violating vertices and apply the changes to them based on a probability. This increases the probability of searching more areas of the search space causing the graph to become *k(d)-neighborhood-anonymized* faster.

Bad Scenario. In the *bad* scenario, there is no advantage to focus only on violating vertices neighborhoods. So we apply the GA to the whole graph. For this purpose, we select the vertices –selection in GA– from all the vertices in graph to apply the changes.

In comparison to the *good* scenario, the *bad* scenario, in general requires more changes and thus more time to create the *k(d)-neighborhood-anonymized* graph. As we will see in Section 5.3, even in *bad* scenarios, our results are much more efficient in terms of minimum number of changes and hence utility preservation compared to the existing techniques.

Ugly Scenario. In the *ugly* scenario, we again focus on violating vertices in the beginning of the GA process like in *good* scenario but we apply more changes in each step of GA compared to *good* scenario so that the graph becomes *k(d)-neighborhood-anonymized* faster. Again, as we move forward, we select some vertices from other non-violating vertices to increase the probability of searching more areas of the search space.

5.2.2 Initial population

In this step we randomly apply edge switches on the original graph to create a fixed number of chromosomes as the initial population. We present chromosome representation details in section 5.2.6. As we discussed earlier, we create a larger initial population in *bad* scenarios compared to the *good* and *ugly* scenarios.

5.2.3 Fitness function and Selection

For each chromosome, the fitness value – which defines how good a solution the chromosome represents – is computed and the chromosomes are selected for reproduction based on their fitness values. Therefore, first, we need to define a function which computes the distance between the modified graph and the original graph (fitness function) and second, we need a function to compute the distance between the modified graph and the solution of a *k-neighborhood-anonymized* graph (selection function). We define the fitness function as below:

$$fitness(G, \tilde{G}) = \frac{1}{size((E \setminus \tilde{E}) \cup (\tilde{E} \setminus E))} \tag{5.1}$$

Given the original graph , is the set of vertices and is the set of edges in , and the modified graph *$\tilde{G}$($\tilde{V}$,$\tilde{E}$)*, we evaluate the distance between the modified graph and the original graph by computing the number of edges in the union of relative complement of E in Ẽ and relative complement of Ẽ in E. Finally, we consider the inverse of the computed number of different edges so that a graph with higher fitness value has fewer changes. After we compute the fitness values, we use roulette wheel selection so that the chromosomes with a higher fitness value will be more likely to be selected. With this method, in each step of GA we select those chromosomes which need fewer modifications to the original graph. As we discussed earlier, we need to define a selection function as well. We define the selection function as the inverse of the number of vertices in the graph that do not satisfy the *k(d)-neighborhood-anonimity* concept for a given *k* and *d*. Using this selection function, in each step of GA, we select those chromosomes which are closer to the solution.

5.2.4 Crossover and mutation

Crossover and mutation are the two basic processes in GA. Crossover process copies individual strings (also called parent chromosomes) into a tentative new population for genetic operations and mutation is used to preserve the diversity from one generation to the next. Mutation prevents the GA from becoming trapped in a local optima. For crossover, the main function we employ is *edge switch* as follows. Given graph and a pair of edges *(u,v)* $\in E$ and *(w,z)* $\in E$ such that *(u,w)* $\notin E$ and *(v,z)* $\notin E$, we remove edges *(u,v)* and *(w,z)* and we add edges *(u,w)* and *(v,z)* to the graph. Note that *edge switch* can be considered as the process of combining the parent chromosomes where the parents are the chromosome and a copy of itself.

For mutation, we remove/add one or some number of random edges to some chromosomes. Specifically in our GA, first we try to perform *edge switch* for a certain number of times (s in Figure 5.3). If we fail to reach to a solution by applying s edge switches, then we start to remove/add one or some number of random edges to some chromosomes to create the new generation and then we repeat the GA for the new generation. If it is a *good* scenario, we remove/add very small number of edges in each step and if it is a *bad* scenario, we remove/add greater number of edges in each step. To decide whether to add or remove edges, if the selected vertex has a degree higher than average graph degree, we remove an edge while if the vertex degree is lower than average degree, we add an edge.

5.2.5 Checking stopping criteria

GAGA always returns at least one *k(d)-neighborhood-anonymized* graph as the solution by trying to apply minimum number of changes (switches, adds, removes) to the original graph. Therefore, in general we only have one stopping criteria except for invalid cases\inputs i.e. suppose a graph with —V—=n is given and a *k(d)-neighborhood-anonymized* graph is requested for some *k*¿n. The problem has no solution unless we add fake vertices. *GAGA* does not introduce any fake vertices as in some previous works [33, 63], since adding fake vertices often makes the generated graph useless by changing the global structure of the original graph.

5.2.6 Implementation highlights

We implemented *GAGA* in Java. The implementation challenges are as follows.

Chromosomal representation. As discussed earlier, we need to represent the graph in an effective way such that *k(d)-neighborhood-anonymity* concept can be put into action and the d distance neighborhood for each vertex can be easily considered. For this purpose, we represent the graph as a *HashMap* structure where each key represents a vertex of the graph and the value represents the *d-neighborhood* structure around the vertex.

Thresholds and parameters. As we discussed, k and d are the main parameters of *GAGA* which provide the data owners some application-oriented level of control over achieving the desired level of data preservation and anonymization. Besides the k and d parameters, *GAGA* contains other thresholds and parameters used in GA: initial population size, s as the number of maximum edge switches before remove/add one or some number of random edges, ,thresholds T_{pv} and T_u to categorize the scenario of the graph, a parameter to indicate finding local maxima as opposed to the global maximum for scenarios where the user/data owner can tolerate some number of violating vertices. *GAGA* receives the above parameters as the input.

Graph isomorphism test. The problem of graph isomorphism which determines whether two graphs are isomorphic or not is NP [39]. The graph isomorphism tests are frequently conducted in the selection phase of GA. For this purpose, we used the VF2 algorithm introduced in [34] as a (sub)graph isomorphism algorithm with efficient performance specially for large graphs. Since the nature of any isomorphism test is that it takes time, we perform multiple level of prechecks to avoid applying the algorithm as much as possible. As a simple example, when two subgraphs have different number of vertices (or edges), or different degree sequences, we do not apply the VF2 algorithm.

5.3 Experimental Evaluation

In this section, first, we evaluate the effectiveness of *GAGA* against the existing De-Anonymization (DA) attacks using real world graph. Second, we evaluate *GAGA* under various utility metrics and compare the results with the state-of-the-art graph anonymization approaches. Finally, we compare the performance of *GAGA* with work by Zhou and Pei [120]. All the experiments were conducted on a PC running Ubuntu 16.04 with an Intel Core i7-4770 CPU running at 3.4 GHz and 23 GB RAM.

5.3.1 Evaluating *GAGA* against DA attacks

As discussed in Section 5.1, Ji et al. [56] implemented the *SecGraph* tool to conduct analysis and evaluation of existing anonymization techniques. In this subsection, we compare *GAGA* with the state-of-the-art anonymization techniques using *SecGraph* against different DA attacks.

Dataset and DA attacks. We use Facebook friendship network collected from survey participants using the Facebook app [61] consisting of 61 nodes and 270 undirected edges representing the friendship between users. We evaluate the anonymization approaches against the following five practical DA attacks:

1) Narayanan and Shmatikov [79]: They proposed a re-identification algorithm to de-anonymize the graph based on the graph topology. Here the attacker, in addition to having detailed information about a very small number of members of the target network, also has access to the data of another graph (a subgraph from the target graph or another social network). Thus, the power of the attack depends on the level of the attacker's access to auxiliary networks.

2) Srivatsa and Hicks [95]: They presented an approach to re-identify the mobility traces. They used the social network data of the participating users as the auxiliary information. They used heuristic based approach on Distance Vector, Randomized Spanning Trees, and Recursive Subgraph Matching to propagate the DA.

3) Yartseva and Grossglauser [114]: They proposed a simple percolation-based graph matching algorithm that incrementally maps every pair of node with at least r (predefined threshold) neighboring mapped pairs. They also showed that the approach used in [79] has a sharp phase transition in performance as a function of the seed set size. That is, when the seed set size is below a certain threshold, the algorithm fails almost completely. When the number of seeds exceeds the threshold, they achieve a high success rate. This is again consistent with the evaluation of [79] which shows that the power of the attack depends on how large the auxiliary networks are.

4) Korula and Lattanzi [60]: They presented a similar approach to [114] where they use an initial set of links of users across different networks as the seed set and map a pair of users with the most number of neighboring mapped pairs

5) Ji et al. [57]: They proposed two DA attack frameworks, namely De-Anonymization and Adaptive De-Anonymization. The later attack is used to de-anonymize data without the knowledge of the overlap size between the anonymized data and the auxiliary data. In their attack, besides the vertices' local properties, they incorporate global properties as well.

Our evaluation methodology is basically the same as in [56]. We compare *GAGA* with the following anonymization techniques:

Add/Del approach introduced in [67] which adds k randomly chosen edges followed by deletion of other k randomly chosen edges.

Deferentially Private Graph Model (***DP***) proposed in [90], in which a partitioned privacy technique is employed to achieve differential privacy.

K-Degree Anonymization (***KDA***) technique presented in [67], in which some edges are added to or removed from the original graph so that each vertex has at least k-1 other vertices with the same degree.

Random Walk Anonymization (***RW***) approach proposed in [76], where the graph is perturbed with replacing the edges by random walk paths in order to provide link privacy.

t-Means Clustering Algorithm (***t-Means Cluster***) introduced in [101], uses conventional t-Means algorithm to create clusters with size of at least k.

Union-Split Clustering (***Union Cluster***) technique presented in [101], is similar to *t-Means Clustering Algorithm* while cluster centers are not chosen arbitrarily to bypass the variability in clustering results.

We present the results in Table 5.1. The criteria for parameters settings for each anonymization technique are the same to settings as in [56] which follows the same settings in original works. That is for *Union Cluster*, k is the size of each cluster; for *Add/Del*, f is the fraction of edges to be modified; for *KDA*, k is the anonymization parameter indicating the number of similar nodes with respect to degree; for *DP*, ε is the parameter that determines the amount of noises that must be injected into the graph where a larger value of ε means that it is easier to identify the source of the graph structure and hence a lower level of graph privacy is preserved; for *t-Means Cluster*, t is the parameter which shows the minimum size of each cluster; for *RW*, r is the number of steps; and finally, for *GAGA*, k indicatess the number of similar nodes with respect to neighborhood structures and d shows the level of *d-neighborhood*. For DA attacks, we randomly sample a graph with probability *s=80% and s=90%* from the original graph as the auxiliary graph and then we apply the graph anonymization approaches to obtain the anonymized graphs. A larger value for s results in successfully de-anonymizing more users since with a large s, the anonymized graph and the auxiliary graph are likely to have similar structures. We also feed each DA technique 20 pre-identified seed mappings. Then we use the auxiliary graph to de-anonymize the anonymized graph. We use *Union Cluster* as the baseline for our evaluation. For each anonymization technique, *SecGraph* provides the number of successfully de-anonymized users – this number for *Union Cluster* is given in parenthesis. For rest of the techniques, Table 5.1 provides the factor by which number of successfully de-anonymized users is reduced in comparison to the baseline. **Note that *GAGA* is the optimal solution against all DA attacks (bold values in Table 5.1) as for all DA attacks *GAGA* offers the most defense – in fact the number of de-anonymized users is either 0 (perfect defense) or 1 (near perfect) of 50–57. *GAGA* (wih d=3) reduces the de-anonymization rate by at least a factor of 2.7× over the baseline. A factor of ∞ means no user**

DA	s	*Union Cluster*	*Add/Del*	*KDA*	*DP*	*t-Means Cluster*	*RW*	*GAGA*		
		(k=5)	*(f=0.23)*	*(k=5)*	*(ε=10)*	*(t=5)*	*(r=2)*	*(k=5 , d=1)*	*(k=5 , d=2)*	*(k=5 , d=3)*
Ji et al. [57]	0.8	1 (2 of 42)	1.1×	1.1×	1.2×	1.3×	1.8×	1.5×	2.3×	black2.7× (1 of 57)
	0.9	1 (2 of 37)	1.1×	1.1×	-	1.4×	-	1.4×	2.2×	black3.1× (1 of 57)
Korula and Lattanzi [60]	0.8	1 (2 of 48)	1.2×	1.2×	1.2×	1.2×	1.3×	1.5×	black∞	black∞ (0 of 51)
	0.9	1 (2 of 45)	0.9×	1.2×	-	1.1×	-	1.2×	2.5×	black∞ (0 of 50)
Narayanan and Shmatikov [79]	0.8	1 (3 of 51)	0.8×	1×	1×	1.6×	1.4×	1.5×	black3.1×	black3.1× (1 of 53)
	0.9	1 (3 of 44)	0.8×	1×	-	1.5×	-	1.5×	3.3×	black3.6× (1 of 53)
Srivatsa and Hicks [95]	0.8	1 (2 of 42)	1.1 ×	1.1×	1.2×	1.3×	1.6×	1.5×	1.9×	black2.7× (1 of 57)
	0.9	1 (2 of 38)	1×	1.1×	-	1.3×	-	1.4×	1.9×	black3× (1 of 57)
Yartseva and Grossglauser [114]	0.8	1 (4 of 52)	1.4×	1.7×	1.8×	2×	2.2×	2.1×	3.6×	black4× (1 of 52)
	0.9	1 (4 of 44)	1.3×	1.5×	-	1.7×	-	2×	3.9×	black4.7× (1 of 52)

Table 5.1: Comparing *GAGA*'s preservation of privacy with existing approaches introduced in [56] against five DA attacks. Using Union Cluster as the baseline, the factor by which number of de-anonymized users is reduced by each other technique is presented.

has been de-anonymized successfully. Larger values of d make *GAGA* more powerful against DA attacks. This is because each DA attack uses a combination of structural properties/semantics while each anonymization technique usually focuses on one structural property/semantic (e.g., vertex degree in *KDA* [67] or *1-neighborhood-anonymity* in [120]). However in *GAGA* we use *k(d)-neighborhood-anonymity* for any *d-neighborhood* which makes all complex neighborhoods structures similar to at least *k-1* other neighborhoods structures followed by other structural properties/semantics changes. Note that no values for *DP* and *RW* when *s=90%* are given because the anonymized graphs obtained in these two cases do not have enough edges; however, we are able to report the results for them when *s=80%*.

5.3.2 Evaluating *GAGA* for Utilities

Now we compare *GAGA* with the state-of-the-art anonymization techniques using *SecGraph* from the graph and application utility preservation perspective.

Dataset and utility metrics. We use DBLP co-authorship network [61] consisting of 8734 nodes and 10100 undirected edges representing the co-authorship where two authors are connected if they publish at least one paper together. We apply the same graph anonymization approaches that we used in previous subsection along with $GAGA$ to anonymize the original graph and then measure how each data utility is preserved in the anonymized graph compared to the original graph. We use the following 16 popular graph and application utility metrics to measure the utility preservation:

Authorities Score: which is the sum of the scores of the hubs of all of the vertex predecessors.

Betweenness Centrality: which indicates the centrality of a vertex. It is equal to the number of shortest paths from all vertices to all others that go through the specific vertex.

Closeness Centrality: which is defined as the inverse of the average distance to all accessible vertices.

Community Detection: a communication in a graph is a set of vertices where there are more connections between the members of the set than the members to the rest of the graph. *SecGraph* uses the hierarchical agglomeration algorithm introduced in [113] to measure the *Community Detection*.

Degree: which indicates the degree distribution of the graph.

Effective Diameter: which is the minimum number of hops in which some fraction (say, 90%) of all connected pairs of vertices can reach each other.

EigenVector: let A be the adjacency matrix of a graph G, the *EigenVector* is a non-zero vector v such that $Av = \lambda v$, where λ is a scalar multiplier.

Hubs Score: which is the sum of the authorities scores of all of the vertex successors.

Infectiousness: which measures the number of users infected by a disease in a infectious diseases spreading model where each user transmits the disease to its neighbors with some infection rate.

Joint Degree: which indicates the joint degree distribution of the graph.

Local Clustering Coefficient: which quantifies how close the vertex neighbors are to being a complete graph.

Network Constraint: which measures the extent to which a network is directly or indirectly concentrated in a single contact.

Network Resilience: which is the number of vertices in the largest connected cluster when vertices are removed from the graph in the degree decreasing order.

Page Rank: which computes the ranking of the vertices in the graph.

Role Extraction: which automatically determines the underlying roles in the graph and assigns a mixed-membership of the roles to each vertex to summarize the behavior of the vertices. *SecGraph* uses the approach in [47] to measure the *Role Extraction.*

Secure Routing: to address the security vulnerabilities of P2P systems, Marti et al. [74] proposed an algorithm to leverage trust relationships given by social links. *SecGraph* uses their approach to measure the *Secure Routing* utility metric.

Table 5.2 presents the results and provides the parameters that were used for each approach. Each value in the table represents one of the the following: *Cosine Similarity* in case of Authorities Score, Betweenness Centrality, Closeness Centrality, Degree, Hubs Score, Infectiousness, Joint Degree, Local Clustering Coefficient, Network Constraint, Network

Utility	***Add/Del*** *(f=0.06)*	***DP*** *(ε=10)*	***GAGA*** *(k=5, d=1)*	***KDA*** *(k=5)*	***RW*** *(r=2)*	***t-Means Cluster*** *(t=5)*	***Union Cluster*** *(k=5)*
Authorities Score	0.4995	0.324	**0.7079**	0.4013	0.3792	0.6773	0.6976
Betweenness Centrality	0.8256	0.762	**0.9606**	0.8459	0.8247	0.9378	0.8755
Closeness Centrality	0.9123	0.785	0.9604	0.9788	0.8612	0.9632	0.9832
Community Detection	0.3926	0.1766	0.8783	0.8747	0.2933	0.5324	0.9103
Degree	0.9877	0.9265	**0.9998**	0.9979	0.9648	0.9972	0.9989
Effective Diameter	0.9559	0.6268	**0.9632**	0.9205	1.7626	0.9394	0.9629
EigenVector	0.8562	0.4443	**0.9927**	0.6573	0.5573	0.9598	0.9909
Hubs Score	0.6844	0.2971	**0.7259**	0.5274	0.3967	0.6997	0.686
Infectiousness	0.8033	0.8675	0.8393	0.8364	0.7093	0.8513	0.8622
Joint Degree	0.7645	0.6102	**0.9875**	0.7713	0.2679	0.6832	0.7943
Local Clustering Coefficient	0.9846	0.9074	**0.9977**	0.997	0.9561	0.9909	0.9939
Network Constraint	0.9885	0.9777	0.9992	0.9999	0.987	0.9994	0.9999
Network Resilience	0.9989	0.9954	0.9997	0.9999	0.9913	0.9999	0.9999
Page Rank	0.3722	0.3323	**0.3766**	0.3681	0.3625	0.3758	0.3742
Role Extraction	0.5519	0.2271	**0.6685**	0.3134	0.2418	0.5282	0.6248
Secure Routing	1.0346	1.1149	**1.0024**	1.007	0.9571	0.9505	1.1717

Table 5.2: Comparing the utility preservation of *GAGA* with the utility preservation of existing approaches introduced in [56] with respect to various utilities.

Resilience, Page Rank, Role Extraction, and Secure Routing; *Ratios* in case of Effective Diameter and EigenVector; and *Jaccard Similarity* in case of Community Detection between the anonymized and original graphs.

For *GAGA*, we set *k=5* and *d=1* and hence as a result, 297 edge adds and 307 edge removes (including 145 edge switches in total) have been applied to the graph. Accordingly, we set the similar parameters for other approaches so that the number of changes to the original graph can be compared with *GAGA* fairly. For example, we used the same *k=5* and *t=5* for *KDA*, *Union Cluster*, and *t-Means Cluster* accordingly. For *Add/Del*, we set f to 0.06, that is because 297 edge adds, and 307 edge removes in *GAGA* map to 307+297 edge adds/deletes for *Add/Del*. We also used a reasonable value for ε in *DP* that is the same value in original work, as we mentioned earlier larger value of ε means smaller changes to the graph so we set ε to the reasonable value of 10. For *RW*, we set r to the minimum value of 2. In general, our evaluation results are consistent with the results presented in [56]:

most of the graph and application utilities can be partially or conditionally preserved with most anonymization algorithms.

Despite the fact that no anonymization scheme is optimal to preserve all utilities, note that for most of the utilities (11 out of 16 highlighted as bold values in Table 5.2) *GAGA* is the best approach to preserve these utilities. For some other utilities, *Union Cluster* and *KDA* have good performance. However, as we discussed in the previous subsection, *Union Cluster* and *KDA* are very vulnerable to DA attacks. This makes *GAGA* the most efficient practical approach which can preserve most of the utilities and at the same time also defend well against modern DA attacks.

5.3.3 *GAGA* vs. Zhou & Pei [120]

As we discussed in Section 5.1, Zhou and Pei [120] presented the *k-neighborhood-anonymity* model to preserve users' privacy against some neighborhood attacks. They evaluated the anonymization cost of their approach using various data sets generated by the R-MAT graph model [32]. To compare our work with Zhou and Pei's work, we used the same model with the same default parameters to generate the same data sets. Figure 5.5 compares the anonymization cost of *GAGA* with their work. Recall that as discussed earlier, Zhou and Pei [120] only support *1-neighborhood* and only apply edge addition to the original graph. However, *GAGA* supports any d and applies three different changes to the graph: switch, add, and remove. Therefore, to compare the cost of *GAGA* to their approach we use *d=1* and we compute the sum of all edge additions and deletions that *GAGA* applies to the original graph. The results show that in all cases *GAGA* is far more efficient in terms of the anonymization cost (i.e., number of changes to the original graph) than Zhou and Pei's

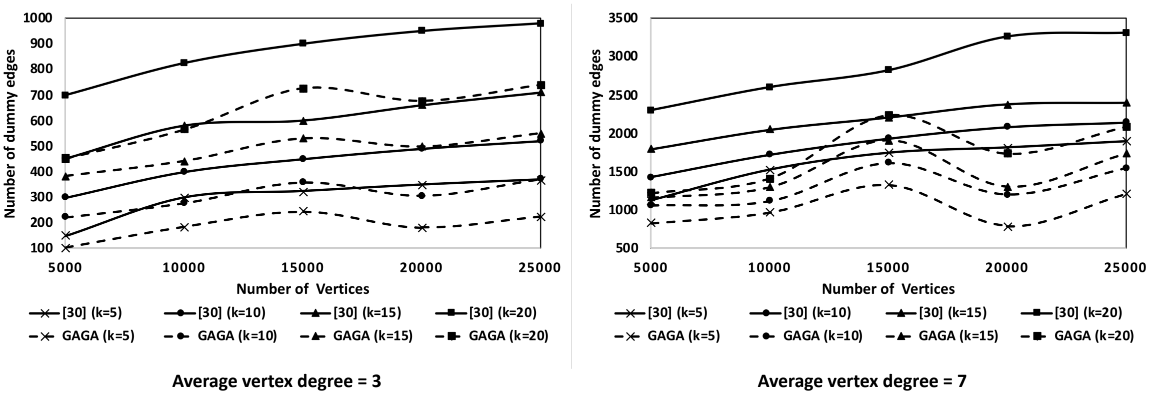


Figure 5.5: Comparing the cost of *GAGA* with the cost of Zhou and Pei [120] on various data sets.

Num. of vertices	k	Average vertex degree=3					Average vertex degree=7				
		Num. of violating vertices (scenario)	Avg. Deg. of violating vertices	Num. of adds	Num. of removes	Avg. (GAGA cost÷Zhou and Pei [120] cost) (%)	Num. of violating vertices (scenario)	Avg. Deg. of violating vertices	Num. of adds	Num. of removes	Avg. (GAGA cost÷Zhou and Pei [120] cost) (%)
5000	5	36(g)	16	39	63	64	321(b)	24	217	614	67
25000	5	116(u)	25	97	129		700(b)	38	388	825	
5000	10	115(b)	14	103	120	72	429(b)	22	315	744	71
25000	10	178(u)	23	177	194		1009(b)	34	457	1093	
5000	15	184(b)	12	175	209	81	505(b)	21	339	827	68
25000	15	284(u)	21	241	309		1187(b)	32	528	1207	
5000	20	217(b)	11	192	261	72	553(b)	20	397	829	61
25000	20	354(u)	19	317	422		1299(b)	32	659	1421	

Table 5.3: *GAGA* anonymization cost on various data sets.

approach when obtaining the same level of privacy preservation. Notice how our approach is efficient even for denser graphs where the average vertex degree is 7 – while the number of dummy edges for Zhou and Pei varies from around 1100 to 3300, the total number of edge adds and removes applied by *GAGA* varies only from 830 to 2230.

We present the results in further detail in Table 5.3. The first column shows the number of vertices used to generate the graphs using R-MAT graph model. For brevity, we report only the cases of 5,000 and 25,000 vertices. The third and eighth column give the number of violating vertices along with the corresponding scenario with respect to different k values (g is the *good* scenario, b is the *bad* scenario, and u is the *ugly* scenario). We give the average degree of violating vertices in fourth and ninth column. A high average degree means that some violating vertices have much higher degree than the graph's average degree (3 or 7) and as a result greater number of removes than adds are needed to anonymize the graph.

Note that since Zhou and Pei [120] only consider *d=1* scenario, the degree of the vertices can be considered as a good parameter to represent the structure of neighborhoods. Thus, we also present the average degree for violating vertices in the tested data sets. Since in *GAGA* we consider *k(d)-neighborhood Anonymization* for any *d-neighborhood*, degree is not a good parameter to represent the complex structure of *d*-neighborhoods. Thus, we report the number of adds, and removes (including edges switches). Finally, we compare the anonymization cost of *GAGA* with Zhou and Pei's [120] cost in the "Avg. (*GAGA* *cost*÷Zhou and Pei [120] cost)" column. **In all cases, our approach is more efficient. On average, our approach incurs only 69% of the cost of Zhou and Pei's approach in terms of number of changes to the original graph**.

5.4 Related Work

As we discussed in section 5.1, several graph anonymization techniques have been proposed. Casas-Roma et al. [31] compare *random-based* algorithm [46] and *k-degree-anonymity* algorithm [67] in terms of graph and risk assessment metrics and it was shown that *k-degree-anonymity* is more effective. The evaluation was limited to 3 small data sets, moreover, only 6 metrics to measure the graph utility preservation are used and no DA attacks were considered in the evaluation. The sole use of degrees in representing graphs and characterizing anonymization introduces limitations. First, it makes anonymization vulnerable to attacks that use more complex graph characteristics such as neighborhood structure of a target vertex. Second, a graph is represented by degree sequence which is not desirable since two different graphs can have same degree sequence.

To overcome the limitations of *k-degree-anonymity*, Zhou and Pei [120] introduced the concept of *k-neighborhood-anonymity* [120] that considers graph structure. As we discussed, they only consider *d=1-neighborhood* which is not efficient for complex DA attacks. Finally, Ji et al. [56] implemented the *SecGraph* tool to analyze existing anonymization techniques in terms of data utility and vulnerability against modern DA attacks. They conclude that it is a big challenge to effectively anonymize graphs with desired data utility preservation and without enabling adversaries to utilize these data utilities to perform modern DA attacks. Therefore, aiming to address the limitations in *k-anonymity* graph anonymization techniques, we implemented and evaluated *GAGA* that not only provides defense against modern DA attacks, but also preserves most of the utilities.

5.5 Summary

We addressed the limitations in graph anonymization techniques. We proposed, implemented, and evaluated *GAGA*, an efficient genetic algorithm for graph anonymization. Our results show that *GAGA* is highly effective and has a better trade-off between anonymization and utility preservation compared to existing techniques. First, by applying the concept of *k(d)-neighborhood Anonymization* for any *d*, *GAGA* preserves data privacy against the modern DA attacks. Second, with the help of genetic algorithm and giving higher priority to edge switching over edge adding and removing, $GAGA$ preserves the graph and application utilities. $GAGA$ gives application-oriented level of control on anonymization and utility preservation to the users via selection of k and d parameters. There are other parameters and thresholds (*GA initial population*, s, T_{pv}, T_u, etc) used in $GAGA$. These could be further tuned to obtain the optimal solutions for any graph.

Chapter 6

Conclusions and Future Work

Any security vulnerability in software has a large effect on user privacy and security, especially if it is in the domain of mobile apps which have entered into people's lives nowadays. We believe that traditional and manual efforts (i.e., reverse engineering, trial and error, and hacky workarounds), are not sufficient to detect several types of potential vulnerabilities in software. Hence, we stress the necessity of having software analysis approaches to help automate vulnerability detection process. We propose couple of software analysis approaches capable of detecting vulnerabilities in software analyzing/testing phase in general, and security assessment in particular. In this context, this thesis makes the following contributions:

- We identify serious security related discrepancies between android apps and their corresponding website counterparts. That is, for the same service, even though the websites are generally built with good security measures, the mobile app counterparts often have weaker or non-existent security measures. As a result, the security of the overall service is only as good as the weakest link.

- We present, implement, and evaluate AndroidSlicer, a novel slicing approach and tool for Android that addresses challenges of event-based model and unique traits of the Android platform. Our asynchronous slicing approach that is precise yet low-overhead, overcomes the challenges and is effective and efficient.

- Upon the application of slicing in mobile apps, we present a new type of vulnerability in identifying app installations on unique devices in Android using dynamic slicing.

- ProgressDroid is the first tool to automatically discover missing progress indicator bugs in Android apps using a technique based on program semantic.

- We present *GAGA*, an efficient genetic algorithm for graph anonymization that is highly effective and has a better trade-off between anonymization and utility preservation compared to existing techniques.

We have shown that there is a practical need for conducting several analysis to detect different types of vulnerabilities in software such as mobile Apps. We demonstrated various applications that can directly benefit from our work. Nevertheless, a wide range of possibilities exists to continue further the research works presented in this dissertation. In this chapter, we outline some future directions.

6.1 Vulnerabilities/Bugs in Mobile Apps

Automatic vulnerability and bug detection techniques presented in this dissertation can be further extended by adding more functionalities to the tools or exploring other potential vulnerabilities.

Dynamic slicing for Android Apps. In Chapter 3, we introduced AndroidSlicer as the first novel slicing infrastructure for smartphone apps. We expect AndroidSlicer will provide a solid foundation for future research. A wide range of applications in security, program verification and debugging, software maintenance and testing, etc can be supported, including *improving dynamic taint analysis*, *undo computing* to detect malicious/buggy actions and restore the system to the last "clean" state, *analyzing ripple effort*, etc.

Progress indicators for long-running operations. In Chapter 4, we described a novel technique for discovering missing progress indicator bugs based on program semantics, in particular program dependencies. Our hope is that our semantic definition of these bugs and our implemented tool will lead to more future work on addressing this type of user interface issues. ProgressDroid automatically detects missing progress indicators for long-running network operations. However, our technique can be extended further to capture missing g progress indicators for any long-running operations, e.g., any types of I/O operations by adding the corresponding API calls. Another interesting future work in this domain would be an approach which can recommend the minimum number of progress indicator handlers needed to cover the long-running operations.

Vulnerabilities in identifying unique devices. In Chapter 4, we presented another new type of vulnerability in techniques that aim to identify unique devices in Android apps. However, other types of related vulnerability can also be taken into consideration. For example nowadays, there are many apps that offer various types of discounts for new users.

These apps use some types of unique device identification mechanisms. Any vulnerability in their mechanisms which might lead to fake activations can cause serious financial damage.

6.2 Preserving User Privacy in Graph Data

In Chapter 5, we introduced, implemented and evaluated $GAGA$ that simultaneously delivers high anonymization and utility preservation. We implemented $GAGA$ via an efficient genetic algorithm. As we showed, there are tuneable parameters and thresholds (s, T_pv, T_u, etc.) in $GAGA$ that impact the performance and runtime of the tool. This opens future directions and challenges to find the efficient values for different parameters to obtain the optimal solutions for any graph.

Bibliography

[1] Best practices for unique identifiers. https://developer.android.com/training/articles/user-data-ids. Accessed:2019-08-08.

[2] jslice, November 2017. http://jslice.sourceforge.net/.

[3] Android api documentation: StrictMode. https://developer.android.com/reference/android/os/StrictMode.html, 2019. Accessed: 2019-08-21.

[4] Android Developer Guides: Keeping your app responsive. https://developer.android.com/training/articles/perf-anr, 2019. Accessed: 2019-08-21.

[5] Progress Indicators - iOS Human Interface Guidelines. https://developer.apple.com/design/human-interface-guidelines/ios/controls/progress-indicators/, 2019. Accessed: 2019-08-02.

[6] Progress indicators - Material Design. https://material.io/design/components/progress-indicators.html, 2019. Accessed: 2019-08-02.

[7] FFmpeg. https://ffmpeg.org/, Retrieved on 02/02/2019.

[8] The Hacker News. Warning: 18,000 android apps contains code that spy on your text messages. http://thehackernews.com/2015/10/android-apps-steal-sms.html, Retrieved on 10/11/2016.

[9] Authentication Policy Table. http://www.cs.ucr.edu/~aalav003/authtable.html, Retrieved on 10/11/2016.

[10] Hacker Selling 200 Million Yahoo Accounts On Dark Web. http://thehackernews.com/2016/08/hack-yahoo-account.html, Retrieved on 10/11/2016.

[11] Red Hat Bugzilla Bug 1204676. https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2015-2331, Retrieved on 10/11/2016.

[12] Charu C. Aggarwal and Philip S. Yu. *Privacy-Preserving Data Mining: Models and Algorithms*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[13] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 246–256, 1990.

[14] Hiralal Agrawal, Joseph R. Horgan, Edward W. Krauser, and Saul London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance*, ICSM '93, pages 348–357, Washington, DC, USA, 1993. IEEE Computer Society.

[15] Abdulmajeed Alameer, Paul T. Chiou, and William G. J. Halfond. Efficiently repairing internationalization presentation failures by solving layout constraints. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 172–182, 2019.

[16] Arash Alavi, Rajiv Gupta, and Zhiyun Qian. When the attacker knows a lot: The gaga graph anonymizer. In *Information Security*, pages 211–230, Cham, 2019. Springer International Publishing.

[17] Arash Alavi, Alan Quach, Hang Zhang, Bryan Marsh, Farhan Ul Haq, Zhiyun Qian, Long Lu, and Rajiv Gupta. Where is the weakest link? a study on security discrepancies between android apps and their website counterparts. In *Passive and Active Measurement*, pages 100–112, Cham, 2017. Springer International Publishing.

[18] Amber. Some Best Practices for Web App Authentication. `http://codingkilledthecat.wordpress.com/2012/09/04/some-best-practices-for-web-app-authentication/`, Retrieved on 10/11/2016.

[19] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, January 2002.

[20] Android Developers. App Components, 2017. `https://developer.android.com/guide/components/index.html`.

[21] Android Developers. UI/Application Exerciser Monkey, November 2017. `http://developer.android.com/tools/help/monkey.html`.

[22] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, June 2014.

[23] Maurizio Atzori. Weak k-anonymity: A low-distortion model for protecting privacy. In *Proceedings of the 9th International Conference on Information Security*, ISC'06, pages 60–71, Berlin, Heidelberg, 2006. Springer.

[24] Tanzirul Azim, Arash Alavi, Iulian Neamtiu, and Rajiv Gupta. Dynamic slicing for android. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1154–1164, Piscataway, NJ, USA, 2019. IEEE Press.

[25] R. J. Bayardo and Rakesh Agrawal. Data privacy through optimal k-anonymization. In *21st International Conference on Data Engineering (ICDE'05)*, pages 217–228, April 2005.

[26] Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable race detection for android applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 332–348, New York, NY, USA, 2015. ACM.

[27] D. Binkley, S. Danicic, T. Gyimothy, M. Harman, Akos Kiss, and B. Korel. Theoretical foundations of dynamic program slicing. In *Theoretical Computer Science*, pages 23–41, 2006.

[28] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. Orbs: Language-independent program slicing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014. ACM, 2014.

[29] Richard Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. Discovering flaws in security-focused static analysis tools for android using systematic mutation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1263–1280, Baltimore, MD, 2018. USENIX Association.

[30] Theodore Book, Adam Pridgen, and Dan S. Wallach. Longitudinal analysis of android ad library permissions. *CoRR*, abs/1303.0857, 2013.

[31] Jordi Casas-Roma, Jordi Herrera-Joancomartí, and Vicenç Torra. *Comparing Random-Based and k-Anonymity-Based Algorithms for Graph Anonymization*, pages 197–209. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[32] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. *R-MAT: A Recursive Model for Graph Mining*, pages 442–446.

[33] Sean Chester, Bruce Kapron, Ganesh Ramesh, Gautam Srivastava, Alex Thomo, and Sistla Venkatesh. Why waldo befriended the dummy? k-anonymization of social networks with pseudo-nodes. *Social Network Analysis and Mining*, 3, 09 2012.

[34] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1367–1372, Oct 2004.

[35] Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. Secsess: Keeping your session tucked away in your browser. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, 2015.

[36] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2187–2200, New York, NY, USA, 2017. ACM.

[37] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *ACM CCS*, 2012.

[38] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[39] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[40] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks . In *2014 Network and Distributed System Security (NDSS '14)*, San Diego, February 2014.

[41] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *ICSE*, 2013.

[42] Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. Java UI : Effects for controlling UI object access. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, pages 179–204, 2013.

[43] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *WiSeC*, 2012.

[44] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. Practical GUI testing of android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 269–280, 2019.

[45] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Proceedings Conference on Software Maintenance 1992*, pages 299–308, Nov 1992.

[46] Michael Hay, Gerome Miklau, David Jensen, Philipp Weis, and Siddharth Srivastava. Anonymizing social networks. Technical report, SCIENCE, 2007.

[47] Keith Henderson, Brian Gallagher, Tina Eliassi-Rad, Hanghang Tong, Sugato Basu, Leman Akoglu, Danai Koutra, Christos Faloutsos, and Lei Li. Rolx: Structural role extraction & mining in large graphs. In *SIGKDD*, pages 1231–1239, New York, NY, USA, 2012. ACM.

[48] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing droids: Program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1844–1851, New York, NY, USA, 2013.

[49] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *ACM Transactions on Programming Languages and Systems*, pages 26–61, 1990.

[50] Susan Horwitz, Thomas W. Reps, and David W. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[51] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336, New York, NY, USA, 2014. ACM.

[52] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. Versatile yet lightweight record-and-replay for android. In *Proc. of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 349–366. ACM, 2015.

[53] Yongjian Hu and Iulian Neamtiu. Static detection of event-based races in android apps. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 257–270, New York, NY, USA, 2018. ACM.

[54] Sungjae Hwang, Sungho Lee, Yongdae Kim, and Sukyoung Ryu. Bittersweet adb: Attacks and defenses. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, 2015.

[55] S. Ji, P. Mittal, and R. Beyah. Graph data anonymization, de-anonymization attacks, and de-anonymizability quantification: A survey. *IEEE Communications Surveys Tutorials*, 19(2):1305–1326, Secondquarter 2017.

[56] Shouling Ji, Weiqing Li, Prateek Mittal, Xin Hu, and Raheem Beyah. Secgraph: A uniform and open-source evaluation system for graph data anonymization and de-anonymization. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 303–318, Berkeley, CA, USA, 2015. USENIX Association.

[57] Shouling Ji, Weiqing Li, Mudhakar Srivatsa, Jing Selena He, and Raheem Beyah. Structure based data de-anonymization of social networks and mobility traces. In Sherman S. M. Chow, Jan Camenisch, Lucas C. K. Hui, and Siu Ming Yiu, editors, *Information Security*, ISC'14.

[58] Yu Kang, Yangfan Zhou, Min Gao, Yixia Sun, and Michael R. Lyu. Experience report: Detecting poor-responsive UI in android applications. In *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*, pages 490–501, 2016.

[59] Bogden Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, October 1988.

[60] Nitish Korula and Silvio Lattanzi. An efficient reconciliation algorithm for social networks. *Proc. VLDB Endow.*, 7(5):377–388, January 2014.

[61] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[62] Christophe Leung, Jingjing Ren, David Choffnes, and Christo Wilson. Should you use the app for that?: Comparing the privacy implications of app- and web-based online services. In *Proceedings of the 2016 ACM on Internet Measurement Conference*, IMC '16, pages 365–372, New York, NY, USA, 2016. ACM.

[63] N. Li, N. Zhang, and S. K. Das. Relationship privacy preservation in publishing online social networks. In *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*, pages 443–450, Oct 2011.

[64] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: A lightweight ui-guided test input generator for android. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 23–26, Piscataway, NJ, USA, 2017. IEEE Press.

[65] Yu Lin, Semih Okur, and Danny Dig. Study and refactoring of android asynchronous programming. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 224–235, Washington, DC, USA, 2015. IEEE Computer Society.

[66] Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, and Gang Wang. Measuring the insecurity of mobile deep links of android. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 953–969, Vancouver, BC, 2017. USENIX Association.

[67] Kun Liu and Evimaria Terzi. Towards identity anonymization on graphs. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 93–106, New York, NY, USA, 2008. ACM.

[68] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[69] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM Trans. Knowl. Discov. Data*, 1(1), March 2007.

[70] Sonai Mahajan, Negarsadat Abolhassani, Phil McMinn, and William G. J. Halfond. Automated repair of mobile friendly problems in web pages. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 140–150, New York, NY, USA, 2018. ACM.

[71] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for android applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 316–325, New York, NY, USA, 2014. ACM.

[72] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 94–105, New York, NY, USA, 2016. ACM.

[73] J. Maras, J. Carlson, and Ivica Crnkovic. Client-side web application slicing. In *ASE'11*.

[74] Sergio Marti, Prasanna Ganesan, and Hector Garcia-molina. Sprout: P2p routing with social networks. volume 3268, 04 2004.

[75] Adam Meyerson and Ryan Williams. On the complexity of optimal k-anonymity. In *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '04, pages 223–228, New York, NY, USA, 2004. ACM.

[76] Prateek Mittal, Charalampos Papamanthou, and Dawn Song. Preserving link privacy in social network based systems. In *NDSS*, 2013.

[77] Greg Mori and Jitendra Malik. Recognizing objects in adversarial clutter: Breaking a visual captcha. In *Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2003.

[78] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian specification learning for finding API usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 151–162, 2017.

[79] A. Narayanan and V. Shmatikov. De-anonymizing social networks. In *2009 30th IEEE Symposium on Security and Privacy*, pages 173–187, May 2009.

[80] Jakob Nielsen. *Usability Engineering.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[81] Jakob Nielsen. 10 usability heuristics for user interface design. `https://www.nngroup.com/articles/ten-usability-heuristics/`, 1994. Accessed: 2019-08-02.

[82] OWASP. Blocking Brute Force Attacks. `http://www.owasp.org/index.php/Blocking_Brute_Force_Attacks`, Retrieved on 10/11/2016.

[83] OWASP. Testing for Captcha (OWASP-AT-012). `http://www.owasp.org/index.php/Testing_for_Captcha_(OWASP-AT-012)`, Retrieved on 10/11/2016.

[84] Pavel Panchekha, Adam T. Geller, Michael D. Ernst, Zachary Tatlock, and Shoaib Kamil. Verifying that web pages have accessible layout. In *Proceedings of the 39th*

ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018, pages 1–14, 2018.

[85] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 925–935, 2012.

[86] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2013.

[87] Filippo Ricca and Paolo Tonella. Construction of the system dependence graph for web application slicing. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '02, pages 123–, 2002.

[88] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. X-pert: A web application testing tool for cross-browser inconsistency detection. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 417–420, New York, NY, USA, 2014. ACM.

[89] Gholamreza Safi, Arman Shahbazian, William G. J. Halfond, and Nenad Medvidovic. Detecting event anomalies in event-based systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 25–37, New York, NY, USA, 2015. ACM.

[90] Alessandra Sala, Xiaohan Zhao, Christo Wilson, Haitao Zheng, and Ben Y. Zhao. Sharing graphs using differentially private graph models. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 81–98, New York, NY, USA, 2011. ACM.

[91] Katie Sherwin. Progress indicators make a slow system less insufferable. `https://www.nngroup.com/articles/progress-indicators/`, 2014. Accessed: 2019-08-02.

[92] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 174–184, New York, NY, USA, 2007. ACM.

[93] Suphannee Sivakorn, Iasonas Polakis, and Angelos D. Keromyti. The cracked cookie jar: Http cookie hijacking and the exposure of private information. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy, 2016*. IEEE, 2016.

[94] Manu Sridharan, Stephen J Fink, and Rastislav Bodík. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[95] Mudhakar Srivatsa and Mike Hicks. Deanonymizing mobility traces: Using social network as a side-channel. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 628–637, New York, NY, USA, 2012. ACM.

[96] Benno Stein, Lazaro Clapp, Manu Sridharan, and Bor-Yuh Evan Chang. Safe stream-based programming with refinement types. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 565–576, 2018.

[97] Attila Szegedi and Tibor Gyimothy. Dynamic slicing of java bytecode programs. *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 35–44, 2005.

[98] Sriraman Tallam, Chen Tian, and Rajiv Gupta. Dynamic slicing of multithreaded programs for race detection. In *ICSM'08*, pages 97–106, 2008.

[99] Sriraman Tallam, Chen Tian, Rajiv Gupta, and Xiangyu Zhang. Enabling tracing of long-running multithreaded programs via dynamic execution reduction. ISSTA '07, pages 207–218, 2007.

[100] Jennifer Tam, Jiri Simsa, Sean Hyde, and Luis V. Ahn. Breaking audio captchas. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 1625–1632. 2008.

[101] Brian Thompson and Danfeng Yao. The union-split algorithm and cluster-based anonymization of social networks. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 218–227, New York, NY, USA, 2009. ACM.

[102] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[103] Paolo Tonella and Filippo Ricca. Web application slicing in presence of dynamic code generation. *Automated Software Engg.*, pages 259–288.

[104] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[105] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[106] Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. Automated layout failure detection for responsive web pages without an explicit oracle. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 192–202, New York, NY, USA, 2017. ACM.

[107] Tao Wang and Abhik Roychoudhury. Dynamic slicing on java bytecode traces. *ACM Trans. Program. Lang. Syst.*, pages 10:1–10:49, 2008.

[108] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 98:98–98:108, New York, NY, USA, 2014. ACM.

[109] Wang, T. and Roychoudhury, A. Using compressed bytecode traces for slicing java programs. 2004.

[110] Dasarath Weeratunge, Xiangyu Zhang, William N. Sumner, and Suresh Jagannathan. Analyzing concurrency bugs using dual slicing. ISSTA'10, pages 253–264, 2010.

[111] T Wolverton. Hackers find new way to milk ebay users. In *Proceedings of the 1998 Network and Distributed System Security Symposium*, 2002.

[112] Jordan Wright. How Browsers Store Your Passwords (and Why You Shouldn't Let Them). `http://raidersec.blogspot.com/2013/06/how-browsers-store-your-passwords-and.html/`, Retrieved on 10/11/2016.

[113] Jaewon Yang and Jure Leskovec. Overlapping community detection at scale: A nonnegative matrix factorization approach. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 587–596, New York, NY, USA, 2013. ACM.

[114] Lyudmila Yartseva and Matthias Grossglauser. On the performance of percolation graph matching. In *Proceedings of the First ACM Conference on Online Social Networks*, COSN '13, pages 119–130, New York, NY, USA, 2013. ACM.

[115] Xiaowei Ying and Xintao Wu. *Randomizing Social Networks: a Spectrum Preserving Approach*, pages 739–750.

[116] Junyuan Zeng, Yangchun Fu, Kenneth A. Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 487–498, 2013.

[117] Hang Zhang, Dongdong She, and Zhiyun Qian. Android root and its providers: A double-edged sword. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1093–1104, New York, NY, USA, 2015. ACM.

[118] Xiangyu Zhang, Sriraman Tallam, and Rajiv Gupta. Dynamic slicing long running programs through execution fast forwarding. SIGSOFT '06/FSE-14, pages 81–91, 2006.

[119] Elena Zheleva and Lise Getoor. Preserving the privacy of sensitive relationships in graph data. In *Proceedings of the 1st ACM SIGKDD International Conference on Privacy, Security, and Trust in KDD*, PinKDD'07, pages 153–171, Berlin, Heidelberg, 2008. Springer-Verlag.

[120] Bin Zhou and Jian Pei. Preserving privacy in social networks against neighborhood attacks. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 506–515, Washington, DC, USA, 2008. IEEE Computer Society.

[121] Bin Zhou and Jian Pei. Preserving privacy in social networks against neighborhood attacks. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 506–515, Washington, DC, USA, 2008. IEEE Computer Society.

[122] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. Harvesting developer credentials in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '15, pages 23:1–23:12, New York, NY, USA, 2015. ACM.

[123] Chaoshun Zuo, Wubing Wang, Rui Wang, and Zhiqiang Lin. Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services. In *NDSS*, 2016.