SPECIFICATION MINING AND AUTOMATED TESTING OF MOBILE APPLICATIONS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Lazaro Clapp
December 2017

Abstract

Static analysis represents a powerful set of techniques for understanding the behavior and security properties of software. However, pure static analysis of large and complex applications is challenging, due both to scalability limitations of precise analyses, as well as the likely presence of dynamic language features such as reflection. We thus propose to combine it with complementary dynamic analysis, which obtains information from actually running the program. A major issue in dynamic analysis involves directing the execution of the program towards all possible behaviors of interest (see e.g. [96, 18]). One way to do this is by obtaining or generating a set of comprehensive program tests or input samples that can drive the dynamic analysis. In this dissertation we examine these issues in the context of application (or "app") frameworks for mobile phones. We examine three distinct research problems.

First, many precise whole-program static analysis techniques run into scalability, precision or soundness limitations when required to analyze the code of the platform itself, due to the large size of the platform's code and dynamic features usually present in this type of code. The standard solution is to construct manual models for the platform behavior. We show that we can mine explicit information flow specifications[1] from concrete executions. Specification mining performs inference from concrete execution data to produce models or specifications of the behavior of code that is outside the scope of our static analysis. In particular, we use a dynamic analysis technique derived from dynamic taint tracking, which "lifts" the taint flows observed to refer only to the arguments and return value of each platform method. These specifications are then consumed by a static analysis system for malware detection, replacing existing manual models. Our technique is able to recover 96.36% of the manual specifications for the static analysis system, which were written over a period of 2 years. It also discovers many more correct annotations that our manual models missed, leading to new end-to-end information flow behaviors being detected by the original static analysis. Although our technique can give rise to false positives, in practice it does so at a slightly lower rate than the rate of manual errors in our hand-written models (99.63% vs 99.55% precision).

This specification mining technique relies on leveraging an existing comprehensive test suite to

[1] Meaning those that capture the transfer of information through data-flow operations executed by the program, rather than implicit information transfer caused by branching in the program's control-flow or other side channels.

obtain sufficient example executions to produce correct specifications. However, in many other cases of interest, the requisite level of test coverage is usually unavailable. This leads us to propose the following two black-box methods for approaching the problem of automated app exploration and testing, which we hope will drive future dynamic analysis techniques.

Second, we present a novel approach for minimizing traces generated by random or recorded UI interactions. This approach is a variant of the classic delta-debugging technique [117, 119], extended to handle application non-determinism. Experimentally, we show that our technique can minimize large GUI event traces reaching particular views of mobile applications, producing traces that are, on average, less than 2% the size of the original traces.

Third, we automate the exploration of mobile applications through an agent that relies exclusively on being able to take screenshots of the application under test, and send input events in response, without need for static analysis or instrumentation. This agent partitions the screen into a grid and keeps track of specific image patches at particular locations in this grid, which cause the application to react when acted upon. An image patch is identified simply by the exact values of every pixel within the small grid square representing the patch location (we use hashing as an optimization). Visual changes to the screen are used as a proxy for application activity. Our tool is able to outperform random GUI testing in method coverage while being robust to a large set of conditions that can easily become limitations for more complex tools.

Our techniques are implemented and evaluated in the popular Android mobile OS. This environment presents significant challenges for static analysis, due to the fact that Android applications are implemented as sets of components embedded into a coordinating runtime and a massive standard API. It also presents an interesting environment for automated testing techniques, due to the large variety of UI toolkits in active use, and the prevalence of application non-determinism.

Acknowledgments

There is an unsurprisingly large number of people who have contributed to keeping me on track, keeping me motivated, and keeping me sane, in the years leading to this dissertation. I mention here only some of them; a set of people certainly necessary, but not sufficient, for me to have made it to this point. Should any of you fail to find your name, knowing that it should have been included, please understand that the listing is well known to be incomplete, despite my best efforts.

First and foremost is, of course, my advisor: Alex Aiken. It goes without saying that without his wisdom, his encouragement, and his patience, none of this would have been possible. Over the last few years, Alex has spent countless hours discussing with me the work presented in this dissertation, and I can say with confidence that more progress was often made in a one hour meeting with him, than in many days of me working at my desk. He has also spent an enormous amount of time helping me revise paper drafts and practice presentations. He taught me much about those skills, on top of all he has taught me of program analysis and computer science. The fact that it was thanks to him that we were all able to remain fed and keep the servers running, should also be noted.

I wish to thank Dawson Engler and Monica Lam as well, not only for being in the reading committee for this dissertation, but for their advice and inspiration during the steps that led to it. Dawson deserves additional gratitude for having been my first year rotation advisor, an appreciation that also goes to John Mitchell. The four above, as well as Trevor Hastie from the Statistics Department, were also in my defense committee, and I wish to thank all five of them once again, as a group, for their time, questions, and insight.

During some portion of my years at Stanford, I had the fortune to sit right next to Saswat Anand, who was a postdoc there at the time. Had this not been the case, not only would a good portion of this work never have happened, but I would have taken much longer to learn many important lessons about how to conduct academic research. Osbert Bastani, another co-author, arrived at Stanford after I did, but taught me no less. Manolis Papadakis and Elliott Slaughter joined the same year I did, and we stayed labmates and friends through rotations, malware detection challenges, dancing lessons and sword fights. I owe them my sanity. Regretting that I lack the space to thank all other inspiring colleagues at Stanford, I limit myself to listing some of those which I got to know best: Patrick Mutchler, Rahul Sharma, Jason Franklin, Stefan Heule, Eric Schkufza, Berkeley Churchill,

Zhihao Jia, Wonchan Lee, Wonyeol Lee, and Pratiksha Thaker. To that set (for it has no given order) should also be added Yu Feng from UT Austin. I thank Sue George (our group's admin), profusely, for her indefatigable efforts in shielding us from the bureaucracy.

Both before and while writing this dissertation, I also had the pleasure of working closely with the following folks in industry: Sam Blackshear, Manu Sridharan, Chenguang Shen and Benno Stein, among many others. The work I did with them might not have ended up in these pages, but I still count myself grateful to have had such extraordinary colleagues.

I also wish to thank those who reminded me, on occasion, that there is life outside the lab window. Within that group, four stand out in both their persistence and their success at making my grad school years significantly more enjoyable than they would otherwise have been: Annie Kwon, Benjamin Prosnitz, Grace Tang, and Madhu Advani. I cannot list all the others without omission, but I shall risk indulging in a very incomplete list: Matthew Ware, Alex Stoll, Huimin Li, Zan Chu, Alex Stoll, Caitlin Bee, Arvind Satyanarayan, Ben Poole, Christie Brandt, Peter Lofgren, Kat Joplin, Ruoxing Liu, Saskia Thiele, Sylvie Bryant, Samata Katta, Yu-Han Chou, Andres Gomez Emilsson. Also, through the screen, Andrea Chapela and Denise Tow.

Defying convention to some degree, I also wish to give thanks, with distinction, to my ex-girlfriend and good friend: Ningxia Zhang. The lows would have been much lower, and the successes less felt, if not for her.

Last, but most certainly not least, I wish to thank my mother, Mónica Clapp. I thank her for the encouragement, for the perspective, for the reassurance, for the living example, for the expert advice on mathematics, and for things far too many to count over almost three decades. I dedicate this dissertation to her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Static analysis is the collective name of a large set of techniques for determining properties of computer programs by analyzing their static structure, as opposed to their runtime behavior during specific executions [33]. This static structure can be the source code of the program or the binary representation itself, as well as any intermediate representation used to convert between the two. In contrast, dynamic analysis techniques look at the actions of the program at runtime. A significant advantage of static analysis compared to dynamic analysis is that it can potentially reason about all feasible executions and can in principle provide soundness guarantees that a program will always satisfy a particular property over all possible executions.

Since static analysis must reason over a potentially unbounded number of executions, it can be difficult to keep the analysis reasonably precise, meaning without a high number of false positives. In general, sound static analysis works by over-approximating the program's possible behavior, as the analysis must account for every possible program behavior that can happen in some execution, and it may include some additional behaviors that can never arise in any execution. Suppose there is a subset of executions exhibiting behavior A, and a subset exhibiting behavior B. If the analysis merges executions of these two subsets in its over-approximation, then it will not be able to rule out the behavior $A + B$ as possible for some execution of the program, even if A and B are, in fact, mutually exclusive. Thus, over-approximation can cause false positives: cases where the analysis cannot prove the absence of violations of a property of interest even in cases where the property in fact holds. The more code the analysis must reason about at a given time, the more limited the compute budget for the analysis, and the more the program makes use of dynamic or polymorphic programming language features (e.g. dynamic typing, reflection or *eval* statements), the less precise the analysis must often become.

Software development in the last two decades has steadily moved from traditional processes running independently directly on top of a classic operating system, to applications developed as sets

of components run by a runtime platform or framework. Within these modern application environments, programs interact with a large base of semantically complex platform APIs and communicate heavily with other applications, as well as with a variety of network-based services for which code is often unavailable for static analysis. Moreover, unlike traditional processes where execution is chiefly controlled by the program's own code, the components of platform-driven applications are directly instantiated and run by the platform itself, based on external system events. Application control flow thus cannot be understood without detailed knowledge of the platform's code.

Examples of this kind of software system range from software built atop MVC (Model View Controller [65, 68]), MVP (Model View Presenter [88]) or MVVM (Model View ViewModel [42]) frameworks, to Apache Struts [40] and Spring [101] applications. In particular, one of the most prominent examples of this mode of development is mobile applications (also called *apps*) running on the Android operating system. The Android OS is the world's most popular mobile operating system, with over 2 billion monthly active users worldwide [87, 77]. As such, it is a premier target of interest for security research (e.g. [32, 113, 38]), general program analysis techniques (e.g. [8, 9, 12]) and automated testing methods (e.g. [74, 3, 53, 15]).

Given the prominence of such platforms, one would like to have available powerful and efficient program analysis tools for applications running on them. However, given the way in which these apps "dissolve" into the platform – both calling into platform code and having their own control-flow be driven by it in turn via callbacks – any whole-program static analysis of platform-driven applications entails analysis of the platform code as well. The case of Android apps and the Android platform is illustrative of the general complexity of this task.

At least four problems make scaling most non-trivial analyses to the Android platform code challenging. First, a very precise analysis may not scale because the code-base in question is very large. Second, Android platform code uses dynamic language features, such as reflection in Java, which are difficult to analyze statically. Third, the platform includes non-code artifacts (e.g., configuration files) that have special semantics that must be modeled for accurate results. Fourth, the platform is built up in layers of abstractions written in lower-level languages, requiring a cross-language analysis framework to carry out a full static analysis of all of the relevant code.

In addition to this hard-to-analyze platform, Android applications often depend heavily on server-side code that posses even greater challenges for static analysis. This code is usually not directly available to an analyst in either source or binary form, and as such it cannot be statically analyzed at all. It can only be treated as missing code to the static analysis.

One popular approach to address the problem of missing code in a static analysis is to use specifications (also called *models*). Writing models is also useful for dealing with hard-to-analyze platform code. A specification substitutes for the missing code and reflects the subset of its effects on the program state that are relevant to the analysis. The analysis can then use these specifications instead of analyzing the platform code. Use of specifications can improve the scalability of an

analysis dramatically because specifications are usually much smaller than the code they specify. In addition to scalability, use of specifications can also improve the precision of the analysis because specifications are also simpler (e.g., no dynamic language features or non-code artifacts) than the corresponding code.

Such specifications are typically written manually in tandem with the construction of a static analysis framework for a given platform. This is a time consuming effort, more so because the specifications must be kept current and updated as the platform changes (as is the case between different versions of the Android OS). As part of this dissertation, we present a technique to mine a particular form of information flow specifications using dynamic analysis over concrete executions of the platform code. To do so, we leverage a pre-existing comprehensive test suite to produce such executions. These specifications match those expected by an existing static taint analysis system for malware detection and improve the performance of that system as a whole when compared with using human-written models.

In general, the idea of using concrete execution data and dynamic analysis to generate platform specifications to be consumed by static analysis is a promising one. However, an important challenge remains in obtaining sufficient example executions of the behavior of interest to be able to mine correct specifications from those executions. In some cases, such as per-method information flow specifications on the Android platform, a complete enough test suite may be available for this purpose. However, in general, a way of systematically producing a set of executions that cover all the relevant behavior for an application, including its usage of the platform, is desired.

We present two contributions to the problem of automated app exploration and testing. We observe that the current state of the art in industry practice for application exploration is variations of "monkey testing" [50]. Monkey tools, and in particular the industry standard tool called simply "Monkey", generate a long sequence of simulated user events sampled from a preset distribution. Over a long enough period of time, this fully random testing is able to explore a significant portion of an app's behavior, and is sometimes compared favorably in practice to more sophisticated automated testing techniques [22]. However, the traces generated by random exploration tend to be very large, with the majority of the events in these traces being irrelevant to any behavior we might care about. Due to their size, such traces are also time consuming to replay and not suitable for test suite construction or specification mining.

We propose a technique, based on delta debugging, for minimizing such traces. An interesting challenge, which our technique overcomes, is that of application GUI-level non-determinism, in which the same application may behave differently under the same sequence of actions in different runs. That is to say, the same action in the same app screen, reached along the same exploration path, might trigger two or more different behaviors with some underlaying probability. Our trace minimization technique deals with this issue by executing candidate traces multiple times during the delta debugging process, and reducing the trace only when the new candidate subtrace triggers

the behavior we care about with high enough probability. We keep the overhead of these multiple executions under control by using one of two scheduling techniques: a custom heuristic and a method based on modeling the candidate subtrace scheduling problem as a Markov Decision Process (MDP). We compare both techniques.

Our second contribution involves an automated exploration agent, which, like Monkey, but unlike the more involved automated testing tools, treats the application under test as a black box. Our agent interacts with the Android application by alternately capturing a screenshot of the running app and producing a new GUI action. The agent tries to determine, for specific actions in specific locations of the screen, given its visual state before and after the action, whether that action triggered any reaction from the app. It uses information about particular clickable and unclickable image patches on the screen, to improve upon random exploration. At the same time, it requires no analysis or instrumentation of the application code, preserving Monkey's robustness to implementation details, such as the programming languages, GUI toolkits, or display technologies, used by the application.

Our thesis is that program analysis of applications running inside a platform framework can be improved by mining specifications for the platform behavior from concrete executions, and that, in general, executions of interest for a dynamic analysis can be produced by automated exploration tools. We present solutions to particular variations of the two sub-problems of specification mining and trace generation, with implementations targeting the Android platform. Where appropriate, we discuss the applicability and limits of execution trace generation and specification mining as general techniques.

1.1 Contributions

As support for our thesis, this dissertation describes one technique for specification mining and two for automated application testing and exploration in detail.

In Chapter 2, we present a technique to mine a particular form of information flow specifications using dynamic analysis over concrete executions of the platform code. These specifications are designed to be consumed by an existing static taint analysis system for malware detection, which we will also briefly describe. We show that, in this setting, our technique is able to recover 96.36% of the specifications previously encoded through extensive manual effort, produce many more specifications that were never captured by manual modeling, and achieve a significant reduction of human effort while having a slightly lower rate of incorrect specifications compared to human-written models.

In Chapter 3, we present a technique based on delta debugging for minimizing large random GUI event traces while preserving some reachable behavior captured by the trace. We evaluate our algorithm on many random traces generated for two sets of commercial and open-source Android applications, showing that we can minimize large event traces reaching a particular application activity. Our approach produces traces that are, on average, less than 2% the size of the original

traces.

In Chapter 4, we present an automated app exploration agent, which relies on learning actionable image patches from screenshots of the application under test. An actionable image patch is a portion of the screen that responds to user actions on it, such as taps in our implementation. We evaluate this agent on a dataset of 15 popular real-world Android applications, taken from the Google Play app store, and show that our agent outperforms random exploration on 9 of those apps and performs at least as well as random testing on the rest. This agent shares the robustness advantages of Android Monkey, as it makes very few assumptions about the implementation of the app under test.

We discuss related work in Chapter 5, and state our conclusions in Chapter 6, briefly describing the larger picture for our approach and possible paths for future research.

1.2 Collaborators and Publications

The work for this dissertation was in collaboration with Alex Aiken, Saswat Anand and Osbert Bastani. A portion of it was only made possible by building on the STAMP static analysis system, developed concurrently with this work by (in alphabetical order): Alex Aiken, Saswat Anand, Osbert Bastani, Bryce Cronkite-Ratcliff, Yu Feng, Jason Franklin, Aravind Machiry, Ravi Mangal, John C. Mitchell, Patrick Mutchler, Mayur Naik, Manolis Papadakis, Rahul Sharma, Xin Zhang, and the author. The ideas discussed here appear in the following conference papers: [24, 25].

Chapter 2

Specification Mining from Concrete Executions

The first contribution presented as part of this thesis is a process for mining platform specifications from concrete executions. In particular, we present a technique which uses dynamic analysis to mine explicit information flow specifications to be used by a static taint analysis system. Both the client static analysis and our mining technique are implemented and evaluated on the Android mobile platform.

We first give an overview of STAMP, the static analysis system that serves as the implementation platform for this work (Section 2.1). We start the section by motivating the use of explicit information flow static analysis as a security tool for filtering potentially malicious or misbehaving mobile applications, as well as the system's requirement for platform specifications and their structure. After a brief foundational discussion of the technologies used to implement STAMP's static analysis components (2.1.1), we briefly describe the details of its points-to (2.1.2) and core explicit taint (2.1.3) analysis components. We then discuss the concrete limitations of our static analysis, particularly as they relate to analyzing code belonging to the Android platform itself (2.1.4). Finally, we revisit how the use of manually written models can help us manage those limitations when analyzing Android apps (2.1.5).

Before we begin describing our specification mining technique, we take another look at those manually written models, present the specification syntax and the particular class of specifications (taint transfer annotations) that we mine automatically (Section 2.2).

We then present the architecture of our dynamic analysis system and discuss the instrumentation strategy (Section 2.3). We proceed by detailing the specification mining algorithm itself, formulated as a special execution semantics to be applied to recorded execution traces of Android platform methods (Section 2.4).

Finally, we present our evaluation and results (Section 2.5). We first compare our mined specifications to existing manually written models for 309 methods across 51 classes, and show that our technique is able to recover 96.36% of these manual specifications and produces many more correct annotations that our manual models missed (2.5.1). Then we show that these new annotations improve the end-to-end results of the STAMP analysis system, allowing it to find new true positive explicit information flow facts about real-world Android applications, compared with what can be obtained using only the manual models (2.5.2). Finally, we control for the quality of the test suite used to produce the executions which drive our specification mining technique, showing that relatively few executions are needed to produce correct specifications for most methods (2.5.3).

2.1 STAMP: Static Taint Analysis for Mobile Programs

As part of a long term research project to improve malware detection techniques on mobile platforms, our research group developed STAMP (see [38]). STAMP is a hybrid static/dynamic program analysis platform for Android applications, combining multiple analysis methods. The core analysis performed by STAMP is a static taint analysis that aims to detect privacy leaks. This is the analysis we will describe in this section, as it is the most relevant to our specification mining technique.

Privacy leaks occur when applications take information from sensitive sources on the user's device and exfiltrate it without permission to external data sinks. For example, an app could scan the phone's contacts list and send it to a remote server without the user's knowledge or permission. Other sensitive sources include: device identifiers, photos and other files, geolocation information, user names and passwords for other apps, among many more. The most common sinks are different types of internet connections. However, the app can also exfiltrate data over SMS, Bluetooth, NFC or any other data connection available on the phone.

The Android OS provides some protection against apps behaving maliciously in the form of access permissions. To read data from a particular source within the device, the app must have been granted the corresponding permission either at installation time or during execution [36, 45]. However, once access has been granted, no controls are enforced on how the app uses information read from that source. This approach has long been considered too limited for practical security, with users often ignoring or failing to understand the implications of permission prompts [37, 62]. Signals based on the category and requested permissions of an app can sometimes be used to heuristically estimate the risk associated with installing a particular application (e.g. [86]). However, many apps have legitimate reasons to access significant private information from the device, but not to exfiltrate it to the outside world. For example, a messaging app needs to be able to read and write to the phone's contact list, and must be able to send messages (either through the Internet or via SMS). Yet there is no compelling reason why the app should be able to send phone number or address information directly from the contact list as a message, especially without user permission.

A better way to evaluate the potential data exfiltration dangers of a particular app would be to produce a list of source-sink pairs. These pairs indicate all the potential combinations in which data can be read from the given source and then subsequently sent to the corresponding sink. Then a human auditor or a learning agent can use that information to decide whether or not the app, based on a description of its use case or a security policy, is doing something unexpected.

A whole-program static taint analysis works by providing a list of sources and sinks to be considered, and then examining the full set of program statements to compute an over-approximation of the potential runtime information flows from sources to sinks. In STAMP's case, specific arguments to Android platform API methods are annotated as sinks, while another set of API methods are annotated so that their return values are considered sources. An argument to a method may also be a source in the particular case in which the annotated method writes sensitive data from the system into the fields of that argument.

The following listing shows the source annotation for the *TelephonyManager.getLine1Number* method, which retrieves the phone number associated with the primary cellphone line of the device:

```
public class TelephonyManager {
    ...
    @STAMP(flows = {@Flow(from=``$PHONE_NUM'',to=``@return'')})
    public String getLine1Number() {
        ...
    }
}
```

Similarly, the next listing shows a sink annotation for the *SocketChannel.write* method, which sends information outside the device through a UNIX socket interface:

```
public class SocketChannel {
    ...
    @STAMP(flows = {@Flow(from=``arg#1'',to=``!INTERNET'')})
    public abstract int write(ByteBuffer src) {
        ...
    }
}
```

As mentioned in the introduction, several issues arise when performing whole-program static analysis on the entire Android platform. We will further discuss these problems in Section 2.1.4, after explaining our particular analysis in detail. To avoid these issues, we also require annotations for API methods that are neither sources nor sinks, but through which information may flow at runtime. Given these annotations, the analysis can restrict itself to looking at the application's own code, and any software libraries included with the app's executable.

For example, the following two annotations correspond to the *CharsetEncoder.encode* and *CharBuffer.put* methods, both of which copy data from their first argument to their return value. In

the second method's case, the return argument is a reference to the *CharBuffer* itself, so we must specify that the data also flows into the corresponding receiver object (*this*):

```
public class CharsetEncoder {
    ...
    @STAMP(flows = {@Flow(from="arg#1",to="@return")})
    public ByteBuffer encode(CharBuffer in) {
        ...
    }
}
...
public class CharBuffer {
    ...
    @STAMP(flows = {@Flow(from="arg#1",to="@return"),
                    @Flow(from="arg#1",to="!this"),
                    @Flow(from="!this",to="@return")})
    public CharBuffer put(String src, int start, int end) {
        ...
    }
}
```

It is easy to see that, for a fully annotated Android platform codebase, annotations of this third type, called *transfer annotations*, are the bulk of the total annotations. Hence, as we will see later in this chapter, we can greatly reduce the manual annotation burden by automatically mining method specifications that subsume transfer annotations. Note, however, that even in the manual case, the annotations need only be written once per version of the Android platform API and can be provided by the developers of tools like STAMP. Individual users of the system do not necessarily need to write any annotations to make use of the tool to analyze new applications.

2.1.1 Datalog and the structure of the core analysis

The core suite of static analyses provided by STAMP is expressed as Datalog programs. Datalog is a logic programming language used to construct deductive databases [106]. A Datalog program consists of relations and rules over relations.

A relation can be conceptualized as a two-dimensional table. The columns are *attributes*, each of which is associated with a *domain*, which defines the set of possible values for that attribute. Rows are tuples of values taken from the domain of the corresponding attributes. If a tuple $(x_0, x_1, ..., x_n)$ is part of a relation R, we say $R(x_0, x_1, ..., x_n)$ is true. An example relation would be Edge(from : Node, to : Node), describing a graph as a set of directed edges between elements of the *Node* domain. A program could then be given an explicit list of edges as input, in a Datalog file such as the following:

$$\begin{aligned} &\text{Edge}(A, B) \\ &\text{Edge}(A, C) \\ &\text{Edge}(B, D) \\ &\text{Edge}(C, D) \\ &\text{Edge}(D, E) \end{aligned} \tag{2.1}$$

A rule in Datalog is given in a prolog-like notation, in which a *head* relation is defined in terms of a set of other relation predicates. Rules might contain concrete values for the elements in a relation (surrounded by quotes), but by default use universally quantified variables instead. Thus, any instance tuple of values which satisfies the right-hand predicates in a rule is taken to satisfy the head relation as well (predicates may be simple relations, as well as negated relations). For example, the following two rules define the *Path* relation in terms of the *Edge* relation and itself, giving the set of all node pairs between which a path exists in the graph:

$$\begin{aligned} &\text{Path}(x, y) :- \text{Edge}(x, y) \\ &\text{Path}(x, z) :- \text{Path}(x, y), \text{Edge}(y, z) \end{aligned} \tag{2.2}$$

The first rule dictates that two nodes are in a *Path* relation whenever they are in an *Edge* relation. The second rule dictates that if any nodes x and y are in a *Path* relation (in that order), and additionally y is in an *Edge* relation with any other node z, then x and z are in a path relation. This is true for any x, y, z satisfying the given subrelations. In particular, following these rules and using the tuples for the *Edge* given in equation 2.1, one can conclude, for example, that $\text{Path}(A, E)$ is true and $\text{Path}(B, C)$ is false.

A Datalog engine is a system that evaluates a Datalog program, answering membership queries for any of the program's relations, including those derived via Datalog rules. Because Datalog rules can produce relations that are far larger than the relations given explicitly as input, it is important for Datalog engines to use a compact representation for relations. STAMP uses the `bddbddb` implementation of Datalog [109], an engine that translates Datalog programs into a representation using BDDs (Binary Decision Diagrams).

We will now describe how Datalog is used to implement STAMP's static points-to and taint analysis algorithms, without focusing too much on how the underlaying engine computes or stores the Datalog relations.

For its core analysis, STAMP reads the bytecode of Android applications (in Android's standard Dex[46] bytecode representation) and produces a series of input relations encoding the individual bytecode instructions which constitute the app. Table 2.1 lists these relations, as well as example

Table 2.1: STAMP's Datalog Input Relations

Relation	Example	Bytecode
Assign(v:V,u:V)	Assign(l1$int@[meth_sig], l2$int@[meth_sig])	i : `l1 = l2`
Alloc(v:V,h:H)	Alloc(l1$Object@[meth_sig], i)	i : `l1 = new Object(...)`
Load(y:V,x:V,f:F)	Load(l1$int@[meth_sig], l2$Object@[meth_sig], f_sig)	i : `l1 = l2.f_sig`
Store(u:V,f:F,v:V)	Store(l1$Object@[meth_sig], f_sig, l2$int@[meth_sig])	i : `l1.f_sig = l2`
LoadStat(y:V,f:F)	Load(l1$int@[meth_sig], f_sig)	i : `l1 = f_sig`
StoreStat(f:F,v:V)	Store(f_sig, l1$int@[meth_sig])	i : `f_sig = l1`
IinvkArg(i:I,n:Z,v:V)	IinvkArg(i, j, a_j) $\forall j$	i : `l1 = meth_sig2(`a_1`, `a_2`, ...)`
IinvkRet(i:I,v:V)	IinvkRet(i, l1$int@[meth_sig])	i : `l1 = meth_sig2(`a_1`, `a_2`, ...)`
MmethArg(m:M,z:Z,v:V)	MmethArg([meth_sig], j, p_j) $\forall j$	i : `public meth_sig(`p_1`, `p_2`, ...)`
MmethRet(m:M,v:V)	MmethRet([meth_sig], l1$int@[meth_sig])	i : `return l1`
chaIM(i:I,m:M)	See below	

elements and the corresponding bytecode. The domains being used are the following:

- Z: (A program-dependent finite subset of the) Natural numbers.
- M: Program methods. The elements of this domain represent each distinct method signature in the application.
- F: Program fields. As above, this contains all fields in the application.
- V: Program variables. In the case of local variables, the elements of this domain distinguish between similarly named variables across different methods in the program as well as the formal parameters of a method. This domain also includes special elements for the *this* and *return* values of each method.
- I: Invocation instruction. This is an identifier for an individual bytecode statement in the app's code for an specific method callsite.
- H: Object allocation location. This is an identifier for an individual bytecode statement in the app's code for an specific object allocation (*new*) operation.
- C: Context. This represents a bounded abstract context, which we will discuss in more detail in Section 2.1.2.

Encoding the basic instructions that make up the program into Datalog relations is the first step for many Datalog-based static analyses. The specific encoding depends on the properties of the code that are relevant for our analysis. For example, Table 2.1 omits the encoding of certain language features, such as control-flow instructions, since our points-to analysis does not make use of intra-procedural control-flow information. In other words, we present a flow-insensitive analysis.

The Assign relation encodes every assignment statement in the program, while Alloc does the same for object allocation. The Load and Store relations deal with object field load and store operations in Java, while LoadStat and StoreStat do the same for operations involving static fields.

IinvkArg and IinvkRet record information about the actual arguments and return value of a method at its call sites, while MmethArg and MmethRet do the same for the binding of the formal parameters inside the callee and every local used in a return statement. For simplicity, we omit rules handling Java primitive values in method arguments and return values, which are similar but are actually distinguished in the STAMP analyses, due to handling of Java's call-by-value semantics for primitive types. This handling is straightforward. The final relation listed, chaIM, is a bootstrapping callgraph analysis based on class hierarchy information. As we will see in the next section, pointer analysis and callgraph construction are intertwined. However, it is important to begin with some reasonable over-approximation of the possible methods called at each callsite. We generate this first coarse approximation by looking at the signature of the method at callsite i and simply adding chaIM(i,m) for every method m which could satisfy the signature (i.e. its receiver is a subtype of the static type on which the method is dispatched at the callsite).

2.1.2 Points to analysis

STAMP uses a field- and context-sensitive Andersen-style pointer analysis [7].

The general idea behind context-sensitivity is that a precise analysis should take into account the context in which a method m is being executed to compute points-to information, because the points-to information varies with the context. Here the context includes information such as the call-site for the different methods in the call stack at the time m is being executed and, for non-static methods, the receiver object of those methods. To understand why this is important, consider the following example:

```
1     public static Object identity(Object o) {
2         return o;
3     }
4     ...
5     public static void main() {
6         Object o1 = new Object();
7         Object o2 = new Object();
8         Object o3 = identity(o1);
9         Object o4 = identity(o2);
10    }
```

Suppose we are working with a context-insensitive analysis, meaning one that assigns a single set of heap locations to each variable in the program's text. This analysis will generate false positive points-to facts for the code above, due to merging information from the two calls to the *identity* function, in two different contexts.

We denote by $v \hookrightarrow h$ the fact that the analysis believes variable v may point to heap object h, or, in Datalog notation: $pt(v, h)$. In particular, let us name the two new heap locations created by the two allocation statements in the code above h_{o1} and h_{o2}, in order of appearance (note that,

regardless of the specifics of the analysis, *main* executes only once, so exactly two heap allocations take place in this example program). Thus, after line 7 of the *main* method, $o1 \hookrightarrow h_{o1}$ and $o2 \hookrightarrow h_{o2}$.

In a typical context-insensitive analysis, there is no way to distinguish different calls to the *identity* method in the code above, so the argument o of this method will be assigned the union of all points-to sets of every variable used in each call to the method. When the method return value is examined, it will carry this inflated points-to set, creating spurious points-to facts for $o3$ and $o4$.

In particular, after the first call in line 8, the argument o of this method will be recorded as potentially pointing to h_{o1} and we will have $o \hookrightarrow h_{o1}$ and $o3 \hookrightarrow h_{o1}$. Now, for the second call to *identity*, the analysis will add the fact that o may point to h_{o2}, and then propagate the points-to set of o into $o4$, resulting in $o4 \hookrightarrow h_{o1}$ and $o4 \hookrightarrow h_{o2}$. Clearly, the first of those two points-to facts is spurious, since when called from line 9, *identity* will never receive something pointing to h_{o1} and thus can't return a reference to h_{o1}. Yet, it is impossible to reconcile this fact with the requirement of assigning to argument o in method *identity* a unique points-to set. In fact, if the points-to analysis is also flow-insensitive (meaning, roughly, that it ignores the order in which statements may execute, by disregarding the control-flow information encoded in the program), then it will also eventually add the spurious points-to "fact" $o3 \hookrightarrow h_{o2}$.

A conceptually trivial way to solve the above imprecision is by creating two copies of the identity method, one for each call-site. Note that the following code can then be analyzed precisely by a context-insensitive analysis:

```
1     public static Object identity_1(Object o) {
2         return o;
3     }
4     ...
5     public static Object identity_2(Object o) {
6         return o;
7     }
8     ...
9     public static void main() {
10        Object o1 = new Object();
11        Object o2 = new Object();
12        Object o3 = identity_1(o1);
13        Object o4 = identity_2(o2);
14    }
```

Of course, extending this approach to a larger program, with many more call-sites for each method and deep copies to resolve nested methods, quickly becomes impractical. However, we can do similarly well by merely keeping track of the context in which the method is being executed, and adding it to our representation of points-to facts. Thus, we represent "variable v may point to heap object h in context c" as $v \hookrightarrow_c h$ or $pt(c, v, h)$. If our context is merely the call-site of the method to which v belongs, that is sufficient to get the following (precise) points-to facts for our original example: $o1 \hookrightarrow h_{o1}$, $o2 \hookrightarrow h_{o2}$, $o \hookrightarrow_{main:8} h_{o1}$, $o3 \hookrightarrow h_{o1}$, $o \hookrightarrow_{main:9} h_{o2}$ and $o4 \hookrightarrow h_{o2}$.

The context can incorporate multiple levels of call-sites to increase the precision of the points-to analysis. However, due to recursion, the full call-stack-based context of a method may not be possible to determine statically from the program text. Additionally, the cost in time and memory of distinguishing a large number of contexts for each variable quickly becomes impractical. Most context-sensitive analyses are properly termed *bounded context-sensitive* analyses, in which only the most recent portion of the call-stack-based context is retained.

STAMP's definition of bounded context-sensitivity is similar to the technique described in [61], combining call-site sensitivity (kCFA) [78] and object-sensitivity based on allocation site [79]. Here, kCFA is the approach we have described so far, in which the context is the k-tail of all call-sites in the stack. The solution to our simple example, in which the two call-sites of the *identity* method produced two different contexts for o, can be properly termed 1-CFA. Object-sensitivity is an approach based on the observation that, in the case of instance methods in object oriented programs, it is often more useful to distinguish calls to method m by its receiver than by its call-site [61]. STAMP takes advantage of this fact by allowing the first element, and only the first element, of the k-tuple representing the context to be an allocation instruction (`new...`) as opposed to a call-site.

Previous to the Datalog analysis, STAMP calculates a relatively imprecise call-graph for the program, using a class hierarchy analysis [29], and populates the context domain C with all feasible k-tuples (by default $k = 2$) representing the possible bounded contexts of every invocation or allocation site in the program. That is, for example, if allocation statement $i : v = newObject()$ and method call $j : r = v.m2(...)$ both occur inside method m, then for each feasible call-site l of m the contexts (l, i) and (l, j) are added to C. Since at this point call-site feasibility is being determined by the coarse class hierarchy analysis, the domain C will likely contain many contexts that are not truly reachable in the analyzed program. STAMP filters those contexts out during the points-to analysis itself, so we only care that the domain encompasses a superset of the reachable contexts. While populating this domain, the analysis also explicitly enumerates the following relations:

- MV(m:M, v:V): v is a variable in method m.
- CC(c:C, d:C): c is the tail of context d, meaning the $(k-1)$ first elements of c are the $(k-1)$ last elements of d. This only makes sense for $k > 1$.
- CI(c:C, i:I): i is a call-site and the first element of context c.
- CH(o:C, h:H): h is an allocation site and the first element of context o.
- HT(h:H, t:T): h is an allocation site for objects of type t.

Figure 2.1 shows a simplified version of the Datalog rules for STAMP's points-to analysis. It omits rules dealing with static methods and fields, missing-code specifications (see the rest of this chapter) and type reachability ($reachableT$).

Figure 2.1: STAMP's Datalog Points-to Analysis

```
1   # Points-to and Field-points-to relations:
2   pt(c,v,o) :- Assign(v,u), pt(c,u,o), typeFilter(v,o).
3   pt(c,v,o) :- Alloc(v,h), MV(m,v), reachableCM(c,m), CC(c,o), CH(o,h).
4   fpt(o1,f,o2) :- pt(c,v,o2), Store(u,f,v), pt(c,u,o1).
5   pt(c,y,o2) :- pt(c,x,o1), Load(y,x,f), fpt(o1,f,o2).
6   pt(c,u,o) :- DVDV(c,u,d,v), pt(d,v,o), ipFilter(c,u,o).
7
8   # Context feasibility and call-graph construction rules:
9   CIC(c,i,d) :- CC(c,d), CI(d,i).
10  DIC(c,i,o) :- IinvkArg(i,0,v), pt(c,v,o).
11  ICM(i,o,m2) :- VirtIM(i,s), Dispatch(t,s,m2), HT(h,t), CH(o,h).
12  CICM(c,i,o,m) :- reachableCI(c,i), DIC(c,i,o), ICM(i,o,m).
13  DVDV(d,u,c,v) :- CICM(c,i,d,m), MV(m,u), param(u,v,i).
14  DVDV(c,u,d,v) :- CICM(c,i,d,m), MV(m,v), return(u,v,i).
15  reachableCM(0,0).   # main method
16  reachableCM(0,m) :- ClinitTM(t,m), reachableT(t).
17  reachableCM(c,m) :- CICM(_,_,c,m).
18  reachableCI(c,i) :- MI(m,i), reachableCM(c,m).
```

For completeness, we briefly describe the rules defining the points-to relation itself. Flow-insensitive, context-insensitive points-to analysis is an extremely well-studied problem and the analysis rules presented here are similar to many of the formulations in the literature [7]. The general idea of these rules is to propagate the points-to sets of variables following the way the values themselves are copied through the program statements, disregarding the control-flow of the program, but keeping track of the contexts we defined previously. Evaluating these rules to a fix-point assigns a points-to set to each variable and context pair that is consistent with what we would observe if we had duplicated the code of each method for every feasible context.

The first rule (line 2) ensures that references are copied by assignment statements, that is: whenever $v = u$ and $u \hookrightarrow_c o$, then $v \hookrightarrow_c o$. Note that abstract heap locations themselves are identified by the full context of their corresponding allocation site (i.e. $o \in C$). The definition of the relation *typeFilter* is omitted here, but its function is pruning some imprecise points-to facts when we know for certain that a particular abstract heap location may not be assigned to v due to type information. The second rule (line 3) handles allocation instructions, by specifying that if $h : v = newX()$ is a statement in method m, then for every valid context c for m, $v \hookrightarrow_c o$, where o is c extended by h. The next two rules (lines 4-5) deal straightforwardly with loads and stores, where the new relation $fpt(o1, f, o2)$ means "field f of abstract heap location $o1$ points-to abstract heap location $o2$". The last rule for the points-to relation (line 6) deals with method arguments and return values. The relation $DVDV(c, u, d, v)$ matches the formal argument u of method m in context c to actual argument v in the calling context d. Thus if a call-site $m_1(..., v, ...)$ in context d may dispatch to method $m(..., u, ...)$ in context c and $v \hookrightarrow_d o$, then $u \hookrightarrow_c o$. Note that the method's identity (m) is implied by the identity of the formal argument u, via the MV relation, and thus omitted from the attributes of $DVDV$ (recall that we assign a unique identifier to every local variable of the

program, without duplicate identifiers across methods). The relation $DVDV(c, u, d, v)$ also matches the formal and actual return values, but with the contexts c and d reversed, so that if $r = m_2(...)$ in context c may dispatch to method m_3 in context d and $v \hookrightarrow_d o$ is a candidate for m_3's actual return value, then $r \hookrightarrow_c o$. The relation *ipFilter* is also a form of type-based assignment filtering over method arguments, similar to the use of *typeFilter* for assignment instructions.

The second half of the program listed in Figure 2.1 gives the rules for constructing the final call-graph, figuring context feasibility and computing the previously mentioned $DVDV$ relation. Note that, in an object oriented language, we must compute the call-graph together with the points-to sets, since the method being called depends on the type of the receiver object due to dynamic dispatch. The core idea for reachability is to begin by considering the main method[1] and the static class initializers of every type being used in the program as reachable. Then, we recursively define relation $CICM(c, i, o, m)$ which says call-site i in context c can invoke method m in context o, but only if (c, i) itself is reachable. $CICM(c, i, o, m)$ decomposes into $DIC(c, i, o)$ which means that the receiver object at call-site i in context c may be the abstract heap context o, and $ICM(i, o, m)$ which checks that the signature (s) of the method invocation at call-site i matches $o.m$. The relations $VirtIM(i, s)$ and $Dispatch(t, s, m2)$ are built by scanning the application bytecode: the first matches call-sites to the method signature of the method being called, whereas the second maps method signatures to concrete methods for specific types t. Given $CICM$, as well as the $param$ and $return$ relations (easily derived from $IinvkArg$, $MmethArg$, $IinvkRet$ and $MmethRet$ from Table 2.1), constructing $DVDV$ is straightforward.

2.1.3 Explicit taint analysis

Given a reasonably precise global points-to analysis, such as the one just described, as well as the source, sink, and transfer annotations introduced at the beginning of this section, the actual implementation of STAMP's taint analysis is straightforward. The core relations used are *tainted(o: C, l: Label)* and *flow(src: Label, sink: Label)*. Here *Label* is the domain of source and sink labels (e.g. "!INTERNET"), extracted from the corresponding source and sink annotations. The *tainted* relation associates abstract heap locations (distinguished up to their points-to analysis context) with zero or more labels. The *flow* relation list pairs of source (*src*) and sink (*sink*) labels, such that a value might flow at runtime from a method argument or return annotated with *src* to a method argument annotated with *sink*.

Figure 2.2 lists the main rules implementing STAMP's taint analysis. Here the *srcLabel*, *sinkLabel* and *transfer* relations are explicitly enumerated based on the Android platform annotations included in STAMP, and the variables listed in them correspond to specific formal arguments (including the

[1] A reader familiar with Android development may note that Android applications do not have a main method, but instead have multiple entry-point callbacks that are triggered by the Android OS's event system. STAMP builds a simple driver for every app, mocking the Android runtime and non-deterministically executing all available entry points. The main method of this driver is used as the single entry-point for the static analysis.

Figure 2.2: Simplified STAMP's Datalog Taint Analysis

```
1  tainted(o,l) :- srcLabel(v,l), pt(c,v,o).
2  tainted(o2,l) :- tainted(o1,l), transfer(v,u), pt(c,v,o1), pt(c,u,o2).
3  tainted(o2,l) :- tainted(o1,l), fpt(o1,_,o2).
4  flow(l1,l2) :- tainted(o,l1), sinkLabel(v,l2), pt(c,v,o).
```

receiver object) or the return value of the annotated methods. The first rule says that if an argument or return value is annotated with a source label l, then the corresponding abstract heap location in the points-to analysis is tainted with that label. We can then rely on the points-to analysis to propagate the taint through the app's code. Such propagation might be broken by passing through a platform method, since our points-to analysis does not analyze platform code. The second rule handles this case, by propagating taint across calls to platform methods, using the transfer annotations. The third rule propagates taint up through field references. The reasoning is that if we ever pass an object to a sink from which any sensitive data can be reached, then we can't know for sure whether or not the API method corresponding to such a sink will read that sensitive data. Finally, the fourth rule detects when a tainted value reaches a method argument that is associated with a sink annotation, producing the corresponding tuple in the *flow* relation.

This version of STAMP's taint analysis is simplified in a few ways compared to the version implemented in practice. First, we omit handling of primitive values and taint transfer for primitive operations, restricting ourselves to the straightforward case of reference values. Second, STAMP attempts to reconstruct feasible "trails" through the code that exemplify specific taint flows, and the bookkeeping relations for this task are somewhat involved. We omit these features as they are mostly out of scope for the work presented here, but note them for completeness.

We should also mention that the type of taint analysis performed by STAMP is *explicit taint analysis*. We track only explicit taint flow, in which the app exfiltrates information through the data flow of the program. A malicious application could also exfiltrate sensitive information through manipulation of a program's control flow, using loop counters or timing-based channels to copy a value without a direct assignment, load or store statement involving the sensitive value. Exploiting control flow to leak sensitive data is known as implicit information flow and is not currently handled by STAMP or any other static malware detection tool of which we are aware.

2.1.4 Limitations of the core analysis

The analysis we have described so far is sound (does not miss any potential explicit taint flows) when dealing with application code that has no missing or reflective method calls, and achieves significant precision (low false-positive rate) when dealing with most applications. However, were we to try to extend this analysis to cover the entirety of the Android platform codebase, instead of using transfer annotations as described in the beginning of Section 2.1, we would quickly run into

several limitations that make it unsuitable for the task.

The analysis we described processes only Java code, which means that all calls to native methods must be treated as calls to missing code. In addition, we do not incorporate any significant string analysis, which would be required to reasonably approximate the potential targets of reflective method calls. The analysis further treats missing code optimistically, meaning it assumes calls to missing code or reflective calls generate no new points-to facts and produce no taint transfer. The alternative, treating missing code pessimistically (i.e. assuming it can generate points-to facts and taint transfer between any abstract heap locations available to it), leads to an extremely large number of false-positives. Since the Android platform makes extensive use of both reflection and native methods, we would need transfer annotations for those uses, at least.

We would also quickly run into scalability issues when analyzing the Java portion of the platform code. First, the Android platform is one or two orders of magnitude larger as a codebase than even the largest apps, so it is likely we would have to significantly tweak our analysis to even get it to run to completion on the entire Android platform. A second and subtler issue is that platform code tends to be highly polymorphic with many levels of indirection for some common operations, making our analysis choice of $k = 2$ call-sensitivity plus potential object-sensitivity not precise enough in many cases. Increasing the level of context-sensitivity in any way would further reduce the scalability of the analysis. Given that, and the fact that models for platform methods need to be updated only in the relatively rare case that a method's interface changes, we chose to put our annotations at the boundary between application and platform code and replace the platform code with our models.

Finally, we should mention two other known limitations of our analysis, which affect platform as well as application code. First, as we mentioned before, by design we detect only explicit taint flows, and miss implicit information flows caused by the program's control flow. This means that adversarial programs can construct flows we don't directly observe in our analysis, using control-flow based side-channels. A number of patterns to do this, such as implicitly copying a string by using a loop containing a switch over the string's alphabet, are easy to detect via complementary pattern-based static analyses, but the general case remains unsolved for our system or any other Android malware detection tool of which we are aware. Second, we do not explicitly discriminate array accesses by their index, nor do we precisely match accesses to other complex data structures such as linked lists or hashmaps. This significantly reduces our precision when dealing with data structures which mix sensitive and non-sensitive data. We have found these problems not to impede practical analysis of Android applications, but would not be surprised to run into issues arising from these limitations if analyzing platform code.

2.1.5 Manually written models

Due to the limitations described in the previous section, STAMP forgoes static analysis of the Android platform's code, relying instead on a set of models written manually. These models are

a combination of simplified code implementations and annotations. The annotations are those described in the beginning of Section 2.1: source, sink, and transfer annotations. The simplified implementations are fragments of Java code, designed not to be run but to produce points-to facts useful for our analysis.

For example, the following "implementations" of *ArrayList.set* and *ArrayList.get* are incorrect as far as a running program is concerned, since instead of adding the value to a multi-element list data-structure, *set* simply overrides a single object field. This same unique field is then returned by *get* regardless of the index being requested. However, from the point of view of a flow-insensitive path-insensitive static points-to analysis as implemented by STAMP, this is equivalent to the real implementation, since we only care to say that, for the same *ArrayList* object, the second argument to its *set* method may point to the return value of its *get* method. This is in addition to the included transfer annotations[2]:

```
1  class ArrayList<E>
2  {
3      private E f;
4
5      @STAMP(flows = {@Flow(from="arg#2",to="this")})
6      public E set(int index, E object) {
7            this.f = object;
8            return this.f;
9      }
10
11     @STAMP(flows = {@Flow(from="this",to="@return")})
12     public E get(int index)
13     {
14           return f;
15     }
16 }
```

Not all methods require annotations or simplified implementations. For example: methods that write no information to their arguments, nor return a value beyond a status code, such as those used by the Android UI toolkit, can cause no explicit taint flows and thus require no specifications. The main API version targeted by STAMP was the Android 4.0.3 platform, which has a total of 46,559 public and protected methods. STAMP includes manual models for 1,116 of those methods, which were written during 2 years of development in a demand-driven way. As we will see in the next sections, automated specification mining can quickly recover most of the models written over the years, surface errors on those manual models, and produce new and useful specifications for many more methods our manual effort had missed.

[2]Which are potentially redundant in this example, but shown for completeness.

```
1  // Set-up objects
2  SocketChannel socket = ...;
3  CharBuffer buffer = ...;
4  CharsetEncoder encoder =
5      Charset.forName("UTF-8").newEncoder();
6  TelephonyManager tMgr = ...;
7  // Leak phone number:
8  String mPhoneNumber = tMgr.getLine1Number();
9  CharBuffer b1 = buffer.put(mPhoneNumber,0,10);
10 ByteBuffer bytebuffer = encoder.encode(b1);
11 socket.write(bytebuffer);
```

Figure 2.3: Leak phone number to Internet

Table 2.2: Specifications for platform methods

TelephonyManager.getLine1Number()	$PHONE_NUM → return
CharBuffer.put(String,int,int)	arg#1 → this this → return arg#1 → return
CharsetEncoder.encode(CharBuffer)	arg#1 → return
SocketChannel.write(ByteBuffer)	arg#1 → !INTERNET

2.2 Platform Specifications for Explicit Information Flow

Consider the example code fragment of Figure 2.3, corresponding to a portion of a (reachable) method inside an Android application. Given that code – as well as the source, sink and transfer annotations shown at the beginning of Section 2.1 – STAMP should be able to detect that sensitive information regarding the device's phone number (from `TelephonyManager.getLine1Number()`) may flow out into the Internet (via `SocketChannel.write`). This flow involves the taint propagating within the application code itself, as well as through two platform methods: `CharBuffer.put` and `CharsetEncoder.encode`, for which we have transfer annotations.

In the code listings given in Section 2.1, we represented the source, sink and transfer annotations directly as Java annotations, which are one of the formats in which STAMP can read such data. In particular, this is the natural format to use when we wish to combine such taint transfer information with code stubs to create maximally-expressive manual models. However, in the rest of this chapter, we will use a different simplified format, which describes only the information flow specifications for platform methods in isolation from any code-stub models. Table 2.2 shows the information flow specifications for the four platform methods involved in the example of Figure 2.3, using this more compact representation. The notation is as follows:

$a \rightarrow b$ indicates that there is a possible flow from a to b. Whatever information was accessible from a before the call is now potentially accessible from b after the call. Recall from Section 2.1.3, that if a is a reference, the information accessible from a includes all objects transitively

reachable through other object references in fields.

this is the instance object for the modeled method.

return is the return value of the method.

arg#i is the i-th positional argument of the method. For a static method, argument indices begin at 0. For instance methods, *arg#0* is an alias for *this* and positional arguments begin with *arg#1*.

$SOURCE is a source of information flow and represents a resource, external to the program, from which the API method reads some sensitive information (e.g. `$CONTACTS`, `$LOCATION`, `$FILE`).

!SINK is an information sink and represents a location outside of the program to which the information flows (e.g. `!INTERNET`, `!FILE`, `!BLUETOOTH`).

The specifications in Table 2.2, are sufficient for STAMP to track the flow of sensitive information from `$PHONE_NUM`—through parameters and return values—to `!INTERNET`, via static analysis of the code in Figure 2.3.

Over a period of two years, we produced a large set of manually-written models. Generating these models was a non-trivial task, as it required running STAMP on various Android applications, discovering that tainted data flowed into platform methods without existing models, figuring out the full set of platform methods involved in breaking the static flow path, and reading the Android documentation before finally writing a model for each missing method. The majority of these models required no code stubs and involved no new source or sink labels, but were instead explicit taint transfer specifications, expressible in the format of Table 2.2 and involving only flows between the parameters (*this*, *arg#i*) and return values of the specified method. In the rest of this chapter we describe a technique for automatically mining such explicit information flow specifications for arbitrary platform methods.

2.3 Overview of our specification mining system

The main contribution presented in this chapter is a process for mining platform specifications from concrete executions. Now that we have described the client system for these specifications and the form of the specifications themselves, we can begin presenting our technique.

Before delving into the specifics of our specification mining algorithm, we briefly describe the architecture of our system, and the steps of the specification mining process. The source code for our implementation and all related artifacts have been made public, and are available at [23].

Our approach works in four stages: First, we instrument the code corresponding to the Android version for which we wish to mine specifications. Second, we run a large test suite on the

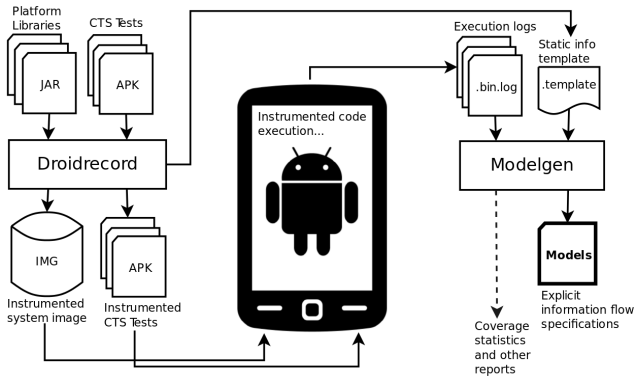


Figure 2.4: Architecture of Droidrecord/Modelgen

instrumented platform, which causes the instrumentation to record execution traces. Third, we process those traces off-line, to produce per-method explicit information-flow specifications. Finally, we transform those specifications into a format equivalent to that of our existing STAMP platform models and annotations.

Figure 2 shows an architecture diagram of our system. The first component, Droidrecord, takes binary libraries (.jar) and application archives (.apk) in their compiled form as Android's DEX bytecode. Droidrecord inserts a special logger class into every executable. Using the Soot Java Optimization Framework [107] with its Dexpler [13] DEX frontend, Droidrecord modifies each method to use this logger to record the results of every bytecode operation performed. We call each such operation an *event* and the sequence of all events in the execution of a program is a *trace*.

Once instrumented, the modified Android libraries are put together into a full Android system image that can be loaded into any standard Android emulator. For specification mining, we capture the traces generated by running the test suite for the platform methods we wish to model. In particular, we use the Android Compatibility Test Suite (CTS) [47]. We consider this kind of test suite as a type of informal specification that is available for many cases of real world systems. Good test suites include examples of method calls that capture the desired structure of the arguments and exercise edge-cases, in a way that, say, executing the method with randomly selected arguments, does not. In later chapters we will explore what can be done in cases where such a comprehensive test suite is not available.

Running the instrumented tests over the instrumented system image produces a collection of *traces*. Each trace consists of an ordered sequence of *events*. Each event in a trace describes a feature of interest in the execution, such as a method call and its arguments (see below). The component of our system that analyzes these traces off-line and generates explicit information flow specifications is called Modelgen, and is the focus of most of the rest of this chapter. However, before describing this analysis, we will briefly describe the instrumentation.

2.3.1 Droidrecord Instrumentation

During the instrumented code's execution, events are recorded to the device (or emulator) file system. On mobile devices this file system often presents limited storage compared to the off-line analysis environment, and thus it is important that the representation of events be compact. A compact representation also reduces the slowdown resulting from performing many additional disk writes as the instrumented code executes. Controlling this slowdown is important, since the Android platform monitors processes for responsiveness and automatically terminates those that take too long to respond [49].

To reduce the amount of data it must record at runtime, Droidrecord first generates a template file (.template) containing all the information for each event that can be determined statically. The instrumented code stores only a number identifying the event in the template file and those values of the event that are only known at runtime. As an example, consider a single method call operation, shown below in Soot's internal bytecode format (slightly edited for brevity):

```
1 $r5 = v_invoke $r4.<StringBuilder.append(int)>(i0);
```

When encountering this instruction, Droidrecord outputs the following event template into its .template file:

```
1 17533:[MethodCallRecord{Thread: _,
2        Name: <java.lang.StringBuilder.append(int)>,
3        At: [...],
4        Parameters: [obj:StringBuilder:_,int:_],
5        ParameterLocals: [$r4,i0],
6        Height: int:_}]
```

The event template includes a unique template identifier, followed by a long-form representation of the event, where statically-unknown values are represented by placeholders. The bytecode is then instrumented to record the identifier, followed by the runtime values of the method's parameters:

```
1 staticinvoke <TraceRecorder.recordEvent(long)>(17533L);
2 staticinvoke <TraceRecorder.writeThreadId()>();
3 staticinvoke <TraceRecorder.writeObjectId(Object)>(r4);
4 staticinvoke <TraceRecorder.write(int)>(i0);
5 $r5 = v_invoke $r4.<StringBuilder.append(int)>(i0);
```

When reading the trace, these values are plugged into the placeholder positions ('_' above) of the event template. For some events (e.g. literal value assignments) all the values can be inferred statically as a simple function of the values of previous events. These events generate event templates but incur no dynamic recording overhead.

We record the following events: new object creation, field load and store operations, method call and return events, as well as thrown and caught exceptions. Additionally, the following bytecode instructions produce event templates during our processing of the app's DEX, but require no instrumentation to be added around them: literal value assignments, variable copying, and direct (arithmetic, boolean, or bitwise) operations. Section 2.4.1 goes into more detail about the structure of our captured traces.

2.3.2 Modelgen Trace Extraction

After tests are run and traces extracted from the emulator, they are first pre-processed and combined with the static information in the .template file. The result is a sequential stream of events for each method invocation; we write $(m : i)$ for the ith invocation of method m. Calls made by $(m : i)$ to other methods are included in this stream, together with all the events and corresponding calls within those other method invocations. Spawning a new thread is an exception: events happening in a different thread are absent from the stream for $(m : i)$, but appear in the streams for enclosing method invocations in the new thread. This separation may break flows that involve operations of multiple threads and is a limitation of our implementation. We did not find any cases where a more precise tracking of explicit information flow across threads would have made a difference in our experimental results.

2.4 Specification Mining

To explain Modelgen's core specification mining algorithm, we describe its behavior on a single *invocation subtrace* $T_{(m:i)}$, which is the sequence of events in the trace corresponding to method invocation $(m : i)$. The events in $T_{(m:i)}$ include the invocation subtraces for all method invocations called from m during invocation $(m : i)$, including any recursive calls to m. We now describe a simplified representation of $T_{(m:i)}$ (Section 2.4.1) and give its natural semantics (Section 2.4.2), that is, the meaning of each event in the subtrace with respect to the original program execution. Modelgen analyzes an invocation subtrace by processing each event in order and updating its own bookkeeping structures. We represent this process with a non-standard semantics: the modeling semantics of the subtrace (Section 2.4.3). After Modelgen finishes scanning $T_{(m:i)}$, interpreting it under the modeling semantics, it saves the resulting specification which can then be combined with the specifications for other invocations of m (Section 2.4.4).

2.4.1 Structure of a Trace

Figure 2.5 gives a grammar for the structure of traces, consisting of a sequence of events. Events refer to constant primitive values, field or method labels, and variables. The symbol $\diamond$ stands for binary operations between primitive values. Objects are represented as records mapping field names to values, which might be either addresses or primitive values. This grammar is similar to that of a 3-address bytecode representing Java operations. However, it represents not static program structure, but the sequence of operations occurring during a concrete program run, leading to the following characteristics:

1. Conditional (`if`, `switch`) and loop (`for`, `while`) operations are omitted and unnecessary; the events in T represent a single path through the program. The predicates inside conditionals

$$
\begin{array}{rll}
T ::= & e^* & \text{(trace)} \\
e \in event ::= & x = pv & \text{(literal load)} \\
| & x = newObj & \text{(new object)} \\
| & x = y & \text{(variable copy)} \\
| & x = y \diamond z & \text{(binary op)} \\
| & x = y.f & \text{(load)} \\
| & x.f = y & \text{(store)} \\
| & x = m(\overline{y}) & \text{(call)} \\
| & return\ x & \text{(return)} \\
| & throw\ x & \text{(throw exception)} \\
| & catch\ x = a & \text{(caught exception)}
\end{array}
$$

$pv \in Primitive\ Value$ $\quad \diamond \in BinOp$
$a \in Address$ $\quad r \in Rec = \{\overline{f : v}\}$
$x, y, z \in Var$ $\quad \rho \in Env : Var \rightharpoonup Value$
$f \in Field$ $\quad h \in Heap : Address \rightharpoonup Rec$
$m \in Method$

Figure 2.5: Structure of a trace

are still evaluated, usually as binary operations.

2. The values of array indices in recorded array accesses are concrete, which allows us to treat array accesses as we would object field loads and stores (e.g., $a[i]$ becomes $a.i$, and note i is a concrete value).

3. For each method call event $x = m_1(\overline{y})$ in $T_{(m:i)}$ there is a unique invocation subtrace of the form $T_{(m_1:j)} = fun(\overline{z})\{var\ \overline{x}; \overline{e}; e_f\}$ where e_f is a return or throw event and $\overline{x}$ is a list of all variable names used locally within the invocation. Again, since we cover only one path through m for each invocation, invocation subtraces may have at most one return event and must end with a return or throw event.

We avoid modeling static fields explicitly by representing them as fields of a singleton object for each class.

2.4.2 Natural Semantics of a Subtrace

Figure 2.6 gives a natural semantics for executing the program path represented by an invocation subtrace. Understanding these standard semantics makes it easier to understand the custom semantics used by Modelgen to mine specifications, which extend the natural semantics. The natural semantics of a subtrace are similar but not identical to those of Java bytecode. The differences arise from the fact that subtrace semantics represent a single execution path.

During subtrace evaluation, an environment ρ maps variable names to values. A heap h maps memory addresses to object records. Given a tuple $\langle h, \rho, e\rangle$ representing event e under heap h and environment ρ, the operator $\downarrow$ represents the evaluation of e in the given context and produces a new tuple $\langle h', \rho'\rangle$ containing a new heap and a new environment. The operator $\overline{\downarrow}$ represents the evaluation of a sequence of events which consists of evaluating each event ($\downarrow$) under the heap and environment resulting from the evaluation of the previous event. The rules in Figure 2.6 describe the behavior of $\downarrow$ and $\overline{\downarrow}$ for different events and their necessary pre-conditions. We omit the rules for handling exceptions since they do not add significant new ideas with respect to our specification

$$\frac{}{\langle h, \rho, x = pv\rangle \downarrow \langle h, \rho[x \to pv]\rangle} \quad \text{(LIT)}$$

$$\frac{a \notin dom(h)}{\langle h, \rho, x = newObj\rangle \downarrow \langle h[a \to \{\}], \rho[x \to a]\rangle} \quad \text{(NEW)}$$

$$\frac{\rho(y) = v}{\langle h, \rho, x = y\rangle \downarrow \langle h, \rho[x \to v]\rangle} \quad \text{(ASSIGN)}$$

$$\frac{\rho(y) = pv_1 \qquad \rho(z) = pv_2 \qquad pv_1 \diamond pv_2 = pv_3}{\langle h, \rho, x = y \diamond z\rangle \downarrow \langle h, \rho[x \to pv_3]\rangle} \quad \text{(BINOP)}$$

$$\frac{\rho(y) = a \qquad h(a) = r \qquad r(f) = v}{\langle h, \rho, x = y.f\rangle \downarrow \langle h, \rho[x \to v]\rangle} \quad \text{(LOAD)}$$

$$\frac{\rho(x) = a \qquad h(a) = r \qquad \rho(y) = v \qquad r' = r[f \to v]}{\langle h, \rho, x.f = y\rangle \downarrow \langle h[a \to r'], \rho\rangle} \quad \text{(STORE)}$$

$$\frac{\begin{array}{c} m = fun(z_1, ..., z_n)\{var\ \overline{x'}; \overline{e}; return\ y'\} \\ \forall i\ \rho(y_i) = v_i \qquad \rho'(y') = v' \\ \langle h, [z_1 \to v_1, ..., z_n \to v_n, \overline{x'} \to undef\,], \overline{e}\rangle \overline{\downarrow} \langle h', \rho'\rangle \end{array}}{\langle h, \rho, x = m(y_1, ..., y_n)\rangle \downarrow \langle h', \rho[x \to v']\rangle} \quad \text{(INV)}$$

$$\frac{\langle h_i, \rho_i, e_i\rangle \downarrow \langle h_{i+1}, \rho_{i+1}\rangle}{\langle h_0, \rho_0, e_0; \ldots; e_{n-1}\rangle \overline{\downarrow} \langle h_n, \rho_n\rangle} \quad \text{(SEQ)}$$

Figure 2.6: Natural semantics

mining technique and exception propagation complicates both the natural and modeling semantics. Our implementation does handle exceptions.

We now consider how the natural semantics represent the evaluation of the following example subtrace fragment which increments a counter at $x.f$:

$$t ::= y = x.f; z = 1; w = y + z; x.f = w$$

Assuming x contains the address a (i.e., $\rho(x) = a$) of heap record $r = \{f : 0\}$ (i.e., $h(a) = r$), LOAD gives us:

$$\langle h, \rho, y = x.f\rangle \downarrow \langle h, \rho[y \to 0]\rangle$$

Applying LIT, BINOP and STORE, respectively, we get:

$$\langle h, \rho[y \to 0], z = 1\rangle \downarrow \langle h, \rho[y \to 0; z \to 1]\rangle$$

$$\langle h, \rho[y \to 0; z \to 1], w = y + z\rangle \downarrow \langle h, \rho[y \to 0; z \to 1; w \to 1]\rangle$$

$$\langle h, \rho[...; w \to 1], x.f = w\rangle \downarrow \langle h[a \to \{f : 1\}], \rho[...; w \to 1]\rangle$$

Using those evaluations for each expression, SEQ gives the full evaluation of the fragment as

$$\langle h, \rho, t\rangle \overline{\downarrow} \langle h[a \to \{f : 1\}], \rho[y \to 0; z \to 1; w \to 1]\rangle$$

$$\frac{l = new\_loc() \qquad c = new\_color()}{\langle h, \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}, \mathbb{D}, x = pv\rangle \downarrow \langle h, \rho[x \to pv], \mathscr{L}[x \to l], \mathbb{C}[l \to \{c\}], \mathbb{G}, \mathbb{D}\rangle} \quad (\textsc{mLIT})$$

$$\frac{a \notin dom(h) \qquad l = new\_loc() \qquad c = new\_color()}{\langle h, \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}, \mathbb{D}, x = newObj\rangle \downarrow \langle h[a \to \{\}], \rho[x \to a], \mathscr{L}[x \to l], \mathbb{C}[l \to \{c\}], \mathbb{G}, \mathbb{D}\rangle} \quad (\textsc{mNEW})$$

$$\frac{\rho(y) = v \qquad \mathscr{L}(y) = l}{\langle h, \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}, \mathbb{D}, x = y\rangle \downarrow \langle h, \rho[x \to v], \mathscr{L}[x \to l], \mathbb{C}, \mathbb{G}, \mathbb{D}\rangle} \quad (\textsc{mASSIGN})$$

$$\frac{\begin{array}{c}\rho(y) = pv_1 \qquad \rho(z) = pv_2 \qquad pv_1 \diamond pv_2 = pv_3 \\ \mathscr{L}(y) = l_1 \qquad \mathscr{L}(z) = l_2 \qquad l_3 = new\_loc() \\ C = \mathbb{C}(l_1) \cup \mathbb{C}(l_2)\end{array}}{\langle h, \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}, \mathbb{D}, x = y \diamond z\rangle \downarrow \langle h, \rho[x \to pv_3], \mathscr{L}[x \to l_3], \mathbb{C}[l_3 \to C], \mathbb{G}, \mathbb{D}\rangle} \quad (\textsc{mBINOP})$$

$$\frac{\begin{array}{c}\rho(y) = a \qquad h(a) = r \qquad r(f) = v \\ \mathscr{L}(a) = l_1 \qquad \mathscr{L}(y, f) = l_2 \\ C = \mathbb{D}(a, f)?\mathbb{C}(l_2) : \mathbb{C}(l_1) \cup \mathbb{C}(l_2)\end{array}}{\langle h, \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}, \mathbb{D}, x = y.f\rangle \downarrow \langle h, \rho[x \to v], \mathscr{L}[x \to l_2], \mathbb{C}[l_2 \to C], \mathbb{G}, \mathbb{D}\rangle} \quad (\textsc{mLOAD})$$

$$\frac{\begin{array}{c}\rho(x) = a \qquad h(a) = r \qquad \rho(y) = v \qquad r' = r[f \to v] \\ \mathscr{L}(y) = l_1 \qquad \mathscr{L}(a) = l_2 \\ \mathbb{G}' = \mathbb{G} + \{c_1 \to c_2 | \forall c_1 \in \mathbb{C}(l_1), c_2 \in \mathbb{C}(l_2)\}\end{array}}{\langle h, \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}, \mathbb{D}, x.f = y\rangle \downarrow \langle h[a \to r'], \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}', \mathbb{D}[(a, f) \to True]\rangle} \quad (\textsc{mSTORE})$$

$$\frac{\begin{array}{c}m = fun(z_1, ..., z_n)\{var\ \overline{x'}; \overline{e}; return\ y'\} \\ \rho_m = [z_1 \to v_1, ..., z_n \to v_n, \overline{x'} \to undef\,] \\ \mathscr{L}_m = \mathscr{L}[z_1 \to \mathscr{L}(y_1), ..., z_n \to \mathscr{L}(y_n), \overline{x'} \to new\_loc()] \\ \langle h, \rho_m, \mathscr{L}_m, \mathbb{C}, \mathbb{G}, \mathbb{D}, \overline{e}\rangle \overline{\downarrow} \langle h', \rho', \mathscr{L}', \mathbb{C}', \mathbb{G}', \mathbb{D}', t\rangle \\ \mathscr{L}'' = \mathscr{L}'[z_1 \to \mathscr{L}(z_1), ..., z_n \to \mathscr{L}(z_n), \overline{x'} \to \mathscr{L}(\overline{x'})] \\ \forall i\ \rho(y_i) = v_i \qquad \rho'(y') = v' \qquad \mathscr{L}(y') = l\end{array}}{\langle h, \rho, \mathscr{L}, \mathbb{C}, \mathbb{G}, \mathbb{D}, x = m(y_1, ..., y_n)\rangle \downarrow \langle h', \rho[x \to v'], \mathscr{L}''[x \to l], \mathbb{C}', \mathbb{G}', \mathbb{D}'\rangle} \quad (\textsc{mINV})$$

$$\frac{\begin{array}{c}\langle h_i, \rho_i, \mathscr{L}_i, \mathbb{C}_i, \mathbb{G}_i, \mathbb{D}_i, e_i\rangle \downarrow \\ \langle h_{i+1}, \rho_{i+1}, \mathscr{L}_{i+1}, \mathbb{C}_{i+1}, \mathbb{G}_{i+1}, \mathbb{D}_{i+1}\rangle\end{array}}{\langle h_0, \rho_0, \mathscr{L}_0, \mathbb{C}_0, \mathbb{G}_0, \mathbb{D}_0, e_0; \ldots; e_{n-1}\rangle \overline{\downarrow} \langle h_n, \rho_n, \mathscr{L}_n, \mathbb{C}_n, \mathbb{G}_n, \mathbb{D}_n\rangle} \quad (\textsc{mSEQ})$$

Figure 2.7: Modeling semantics

where, in addition to some changes to the environment, field f of record r in the heap has been incremented by one.

2.4.3 Modeling Semantics of a Subtrace

To obtain the information flow facts required to construct our specifications, not only are we interested in tracking information flow through the portion of the heap reachable from the arguments and return value of m, but we also want to "lift" these flows so that they refer exclusively to the method arguments and return value rather than intermediate heap locations. We perform both tasks simultaneously, through the modeling semantics of the subtrace. Interpreting the subtrace through its modeling semantics produces, as a side-effect of such interpretation, the specification

giving the explicit information flow relations between the heap subgraphs rooted at each of the method's arguments, as well as its return value.

The modeling semantics augment the natural semantics by associating *colors* with every heap location and primitive value. For subtrace $T_{(m:i)}$, each argument to m is initially assigned a single unique color. The execution of $T_{(m:i)}$ under the modeling semantics preserves the following invariants:

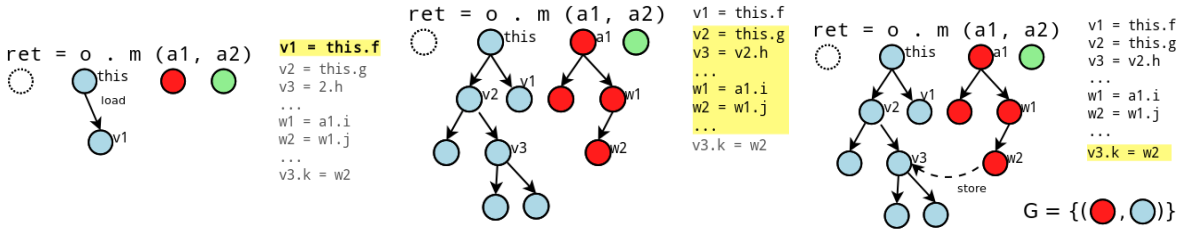
Invariant I: Computed primitive values have all the colors of the argument values used to compute them.

Invariant II: At each point in the trace, if a heap location l is accessed from an argument a using a chain of dereferences that exists at method entry, then l has the color of a.

Invariant III: At each point in the trace, every argument and the return value have all the colors of heap locations reachable from that argument or return value.

These invariants are easily motivated. Invariant I is the standard notion of taint flow: the result of an operation has the taint of the operands. Invariant II captures the granularity of our specifications on entry to a method: all the locations reachable from an argument are part of the taint class associated with that argument (recall the semantics of our specifications described informally in Section 2.2, as well as STAMP's analogous taint propagation along field dereferences from Section 2.1.3). Similarly, Invariant III captures reachability on method exit. For example, if part of the structure of *arg#1* is inserted into the structure reachable from *arg#2* by the execution of the trace, then *arg#2* will have the color of *arg#1* on exit. At every step of the modeling semantics these invariants are preserved for every computed value and heap location seen so far; the invariants need not hold for heap locations and values that have not yet been referenced by any event in the examined portion of the subtrace. In addition, reachability in Invariants II and III applies only to the paths through the heap actually accessed during subtrace execution.

The natural semantics differentiate between primitive values or addresses stored in variables of ρ and objects stored in the heap h. Although this distinction is useful in representing the subtrace's execution, for specification mining we want to associate colors with both heap and primitive values. For uniformity, we introduce a mapping $\mathscr{L}$ which assigns a "virtual location" ($VLoc$) to every variable, object and field based on origin (i.e., where the value was first created) rather than the kind of value. Because virtual locations may be tainted with more than one color (recall Invariant I), we introduce a map $\mathbb{C} : VLoc \rightharpoonup 2^{Color}$ from virtual locations to sets of colors. The modeling semantics also use $\mathbb{G} : \{(Color, Color)\}$, which is a relation on colors or, equivalently, a directed graph in which nodes are colors, and $\mathbb{D} : (Address, Field) \rightharpoonup Boolean$, which stands for "destructively updated" and maps object fields to a boolean value indicating that the field of that location has been written in the current subtrace. At the start of the subtrace's execution, $\mathbb{G}$ is initialized to the empty relation and $\mathbb{D}$ defaults to *False* for all (address, field) pairs. We explain the use of $\mathbb{G}$ and $\mathbb{D}$ below.



(a) Processing a load event

(b) Loads reconstruct the argument's structure

(c) Processing a store event

Figure 2.8: Effects of loads and stores in Modelgen's Modeling Semantics

Figure 2.7 lists the modeling semantics corresponding to the natural semantics in Figure 2.6. We now explain how the first 4 rules preserve Invariant I, as well as how mLoad and mStore preserve Invariants II and III, respectively.

Rule mLit models the assignment of literals to variables. A new literal value is essentially a new information source within the subtrace and is assigned a new location with a new color. The location is associated with the variable now holding the value, preserving Invariant I. Rule mNew, which models new object creation, is similar. Rule mAssign models an assignment $x = y$ where x and y are both variables in ρ and does not create a new location, but instead updates $\mathscr{L}(x)$ to be the location of y, indicating that they are the same value, again preserving Invariant I.

Rule mBinop gives the modeling semantics for binary operations. Assuming locations l_1 and l_2 for the operands, the rule adds a new location l_3 to represent the result. Because of Invariant I, l_3 must be assigned all the colors of l_1 and all the colors of l_2, thus $\mathbb{C}(l_3)$ becomes $\mathbb{C}(l_1) \cup \mathbb{C}(l_2)$.

Rules mLoad and mStore deal with field locations. The virtual location of field $a.f$ (denoted $\mathscr{L}(a, f)$) is defined as either the location of the object stored at $a.f$, if the field is of reference type, or as an identifier which depends on $\mathscr{L}(a)$ and the name of f, if f is of primitive type.

Rule mLoad models load events of the form $x = y.f$ by assigning the location $l_2 = \mathscr{L}(y, f)$ to x and computing the color set for this location (which will be the colors for both x and $y.f$). There are three cases to consider:

- If this is the first time the location $\mathscr{L}(y, f)$ has been referenced within the subtrace $T_{(m:i)}$, then $y.f$ has no color (all heap locations except the arguments start with the empty set of colors in $\mathbb{C}$). Furthermore, since this is the first access, $y.f$ has not been previously written in the subtrace, so $\mathbb{D}(\rho(y), f) = \mathit{False}$. Therefore, l_2 is assigned the colors $\mathbb{C}(l_1) \cup \mathbb{C}(l_2)$ where $l_1 = \mathscr{L}(y)$. Since $\mathbb{C}(l_2) = \emptyset$ before the load event, we end up with $\mathbb{C}(l_2) = \mathbb{C}(l_1)$. If $y.f$ is reachable from a method argument through y, this establishes Invariant II for $y.f$ on its first access.

- If l_2 has been loaded previously in the trace but not previously overwritten, then $\mathbb{C}(l_2) =$

$\mathbb{C}(l_1) \cup \mathbb{C}(l_2)$, indicating that l_2 now has the colors of all of its previous accesses plus a possibly new set of colors $\mathbb{C}(l_1)$. This handles the case where a location is reachable from multiple method arguments and preserves Invariant II.

- If $y.f$ has been written previously then $\mathbb{D}(\rho(y), f) = \mathit{True}$. In this case it is no longer true that $\mathscr{L}(y, f)$ was reachable from $\mathscr{L}(y)$ on method entry and so it is not necessary to propagate the color of $\mathscr{L}(y)$ to $\mathscr{L}(y, f)$ to preserve Invariant II and we omit it. Also, note that if $y.f$ has been written, that implies the value stored in $y.f$ was loaded before the write and so $y.f$ will already have at least one color.

Figure 2.8a shows the effect of a single load operation from an argument to m, while Figure 2.8b depicts the coloring of a set of the heap locations after multiple load events.

Rule mStore models store events of the form $x.f = y$. The rule updates $\mathbb{D}(\rho(x), f) = \mathit{True}$ since it writes to $x.f$. We could satisfy Invariant III by implementing mStore in a way that traverses the heap backwards from x to every argument of m that might reach x and associates every color of y with those arguments (and possibly intermediate heap locations). As an optimization, we instead use $\mathbb{G}$ to record an edge from each color c_1 of $\mathscr{L}(y)$ to each color c_2 of $\mathscr{L}(x.f)$ with the following meaning: $c_1 \to c_2 \in \mathbb{G}$ means every virtual location with color c_2 has color c_1 as well. Figure 2.8c depicts the results of a store operation, while Figure 2.9 depicts how $\mathbb{G}$ serves to associate two colored heap subgraphs.

Rule mInv implements standard method call semantics, mapping the virtual locations of arguments and the return value between caller and callee. Rule mSeq is the same as Seq in the natural semantics, adding $\mathscr{L}$, $\mathbb{C}$, $\mathbb{G}$ and $\mathbb{D}$.

As a consequence of Invariants I and II, the modeling semantics associate the color of each argument to every value and heap location that depends on the argument values on entry to m. Then, because of Invariant III, when the execution reaches the end of subtrace $T_{(m:i)}$ every argument and the return value have all the colors of heap locations reachable from that argument or return value (as represented by $\mathbb{G}$). We construct our specifications by examining the colors of each argument a_j and the return value r after executing the subtrace: for every color of r (or a_j) that corresponds to

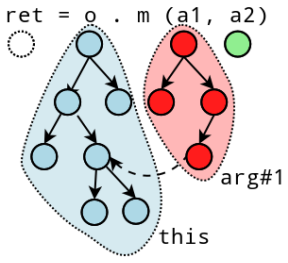


Figure 2.9: Stores connect argument structures

the initial color of a different argument a_k, we add $a_k \to r$ ($a_k \to a_j$) to our model.

2.4.4 Combining Specifications

For each invocation subtrace $T_{(m:i)}$, the process just outlined produces an underapproximation of the specification for m, based on a single execution $(m : i)$. We combine the results from different invocations of m by taking an upper bound on the set of argument-to-argument and argument-to-return flows discovered for every execution, which is simply the union of the results of $(m : i)$ for every i.

For example, consider the method `max(a,b)` designed to return the larger of two numbers, disregarding the smaller one. Suppose that we have two subtraces for this method: one for invocation `max(5,7)`, which returns 7 and produces the model $M_1 = \{arg\#2 \to return\}$ and one for invocation `max(9,2)`, which returns 9 and produces the model $M_2 = \{arg\#1 \to return\}$. Clearly the desired specification reflecting the potential explicit information flow of method `max(a,b)`, with respect to our path- and flow-insensitive client static analysis, is $M_1 \cup M_2 = \{arg\#1 \to return,\ arg\#2 \to return\}$.

We should note that combining specifications in this way inherently introduces some imprecision with respect to the possible flows on a given execution of the method. The effects of this imprecision in our overall system depend on the characteristics of the static analysis that consumes the specifications. For example, the above specification for `max(a,b)` would be strictly less precise than analyzing the corresponding code (assuming the natural implementation) with an ideal path-sensitive analysis, since it merges two different control paths within the max function: one in which the first argument is greater and one in which the second argument is greater. For context-sensitive but path-insensitive analysis such as STAMP (see Section 2.1), loss of precision due to combining specifications is less common, but still possible in theory. Consider a method `do(a,b)` { `a.doImpl(b)` } and two invocations of this method in which `a` has different types and each type has its own implementation of `a.doImpl(b)`. A context-sensitive analysis can tell which version of `doImpl` is executed, but Modelgen simply merges the flows observed for every version of `doImpl` seen in any trace of `do(a,b)`.

2.4.5 Calls to Uninstrumented Code

Our approach to specification mining is based on instrumenting and executing as much of the platform code as we can. Unfortunately recording the execution of every method in the Android platform is challenging. In particular, any technique based on Java bytecode instrumentation cannot capture the behavior of native methods and system calls. Since our inserted recorder class is itself written in Java, we must also exclude from instrumentation some Java classes it depends upon to avoid introducing an infinite recursion. Thus, traces are not always full traces but represent only a part of a program's execution. We need to deal with two problems during event interpretation:

(1) How should Modelgen interpret calls to uninstrumented methods? (2) How can we detect that a trace has called uninstrumented code?

For the first problem, Modelgen offers two separate solutions. The user can provide manually written models for some methods in this smaller uninstrumented subset (as we do, for example, for `System.arraycopy` and `String.concat`). If a user-supplied model is missing for a method, Modelgen assumes a worst-case model in which information flows from every argument of the method to every other argument and to its return value. In many cases, this worst-case model, although imprecise, is good enough to allow us to synthesize precise specifications for its callers. Technically, these "worst-case" models can also be unsound, due to values being persisted in static fields inside the platform code. We didn't find this to be a problem for the platform methods we performed our evaluation on. Generally speaking, this sort of fully global mutable state inside library or platform code is considered an anti-pattern and thus not prevalent. Note that the need for a set of manual models for uninstrumented code does not negate the benefits of Modelgen, since this represents a significantly smaller set of methods. For example, to produce 660 specifications from a subset of the Android CTS (see Section 2.5.1) we needed only 70 base manual models.

The problem of detecting uninstrumented method calls inside traces is surprisingly subtle. Droidrecord writes an event at the beginning of each method and before and after each method call. In the simplest case we would observe these before-call and after-call markers adjacent to each other, allowing us to conclude that we called an uninstrumented method. However, because uninstrumented methods often call other methods which are instrumented, this simple approach is not enough. A call inside instrumented code could be followed by the start of another instrumented method, distinct from the one that is directly called. Dynamic dispatch and complex class hierarchies further complicate determining if the method we see start after a call instruction is the instruction's callee. The canonical example would be a class `B` which inherits from class `A`, and which has an uninstrumented method `B.m(...)` which first calls instrumented method `A.m(...)` and then performs some additional work. If instrumented code calls `a.m(...)` on an object `a` of static type `A`, but dynamic type potentially `B`, then the trace will record a dispatch to `A.m(...)` followed by the start of method `A.m(...)`. Thus, the fact that uninstrumented method `B.m(...)` ran in between those two events would be occluded from us when relying only on the method signatures observed.

Our solution for detecting holes in the trace due to invoking uninstrumented code is to record the height of the call stack at the beginning of every method and before and after each call operation. Since the stack grows for every method call, whether instrumented or not, we use its height to determine when we have called into uninstrumented code. The usual pattern to get the stack height (using a StackTrace object) is expensive. As an optimization, we modify the Dalvik VM to add a shortcut method to get the stack height. Droidrecord will use the shortcut method when available, and gracefully fall back to the more expensive implementation otherwise.

Table 2.3: Comparing Modelgen specifications and manual models

Package	Classes	Methods	Missing trace info.	Total correct flows	Modelgen correct flows	Manual correct flows	Modelgen false positives	Manual errors
java.nio.*	2	26	4	50	50	42	0	0
java.io.*	28	146	23	280	275	234	2	0
java.net.*	7	37	4	104	100	65	0	1
java.util.*	4	28	0	36	36	31	0	1
android.text.*	3	5	2	3	3	3	0	0
android.util.*	2	8	1	11	4	7	0	0
android.location.*	3	13	3	12	12	9	0	0
android.os.*	2	46	3	60	60	49	0	0
Total	51	309	40	556	540	440	2	2

2.5 Evaluation

We perform three studies to evaluate the specifications generated by Modelgen. First, we compare them directly against our existing manually-written models (Section 2.5.1). Second, we contrast the results of running the STAMP static information-flow analysis system using these specifications as input, against the results of the same system using the manual models (Section 2.5.2). Third, we study the effect of test suite quality on the mined specifications (Section 2.5.3).

2.5.1 Comparison Against Manual Models

To evaluate Modelgen's ability to replace the manual effort involved in writing models for STAMP (see Section 2.1 and 2.2), we compare the specifications mined by Modelgen against existing manual models for 309 Android platform methods.

We conducted all of our evaluations on the Android 4.0.3 platform, which has a total of 46,559 public and protected methods. STAMP includes manual models for 1,116 of those methods, of which 335 are inside the `java.lang.*` package which DroidRecord does not currently instrument (this is due partly to performance reasons and partly to our instrumentation code depending on classes in this package, this is not a limitation of the general technique), and 321 have only source or sink annotations, leaving 460 methods for which Modelgen could infer comparable specifications.

For our evaluation, we obtained traces by running tests from the Android Compatibility Test Suite (CTS) [47]. We restricted ourselves to a subset of the CTS purporting to test those classes in the `java.*` and `android.*` packages, but outside of `java.lang.*`, for which we have manual models (not counting simple source or sink annotations). For some packages for which we have manual models, such as `com.google.*`, the CTS contains no tests.

Table 2.3 summarizes our findings, organized by Java package. For each package we list the number of classes and methods for which we have manual specifications, as well as the total number of correct individual flow annotations (e.g. $arg\#X \rightarrow return$) either from our manual specifications or generated by Modelgen. We then list separately the flows discovered by Modelgen and those in our

Table 2.4: Precision and Recall for Modelgen vs Manual Models

Package	Modelgen precision	Manual precision	Modelgen recall
java.nio.*	100%	100%	100%
java.io.*	99.28%	100%	97.86%
java.net.*	100%	98.48%	93.85%
java.util.*	100%	96.88%	100%
android.text.*	100%	100%	100%
android.util.*	100%	100%	0%
android.location.*	100%	100%	100%
android.os.*	100%	100%	100%
Total	99.63%	99.55%	96.36%

manual specifications. We consider only those flows in methods for which we have manual models and only those classes for which we ran any CTS tests, which gives us 309 methods to compare.

We evaluate Modelgen under two metrics: precision and recall. Precision relates to the true positive rate of Modelgen, listing the percentage of Modelgen flow annotations which represent actual possible information flows induced by the method. To determine which Modelgen annotations are correct, we compared them with our manual models and, when the specification for a given method differed between both approaches, we inspected the actual code of the method to see if the differing annotations represented true positives or false positives for either technique. Thus, if $FModelgen$ is the set of flows discovered by Modelgen and TP the set of all flows we verified as true positives, then Modelgen's precision is defined as:

$$PModelgen = |FModelgen \cap TP|/|FModelgen|$$

Similarly the precision of the manual models is:

$$PManual = |FManual \cap TP|/|FManual|$$

Table 2.4 lists the precision of each approach for each package. Both Modelgen specifications and manual models achieve an overall precision of over 99%.

The recall achieved by Modelgen with respect to the manual models is shown in Table 2.4 as well. Recall measures how many of our manual models are also discovered by Modelgen, and is calculated as:

$$Recall = |FModelgen \cap FManual|/|FManual|$$

As we can see, Modelgen finds over 96% of our manual specifications. The specifications Modelgen misses were written to capture implicit flows, which is not surprising since Modelgen is designed to detect only explicit flows. A prime example of this limitation is the row corresponding to `android.util`, in which 7 of the 8 analyzed methods are part of the `android.util.Base64` class,

which performs base64 encoding and decoding of byte buffers via table lookups, inducing implicit flows. Modelgen discovers four new correct flows for these methods, but misses all the implicit flows encoded in the manual models. The last remaining method in this package is a native method.

We can similarly calculate Modelgen's overall recall versus the total number of true positives from both techniques, as well as the analogous metric for Manual:

$$|FModelgen \cap TP|/|TP| = 97.12\%$$

$$|FManual \cap TP|/|TP| = 79.14\%$$

This shows that our technique discovers many additional correct specifications that our manual models missed.

We found two false positives in Modelgen, both in the same method. Two spurious flow annotations were generated, due to a hole in the trace which Modelgen processes under worst-case assumptions (recall Section 2.4.5). Notably, we also found two errors in the manual models: one was a typo (*arg#2* $\rightarrow$ *arg#2* instead of *arg#2* $\rightarrow$ *return*) and the other was a reversed annotation (*arg#1* $\rightarrow$ *this* instead of *this* $\rightarrow$ *arg#1*).

Our current implementation of Modelgen failed to produce traces for a few methods that have manual annotations, listed under the column "Missing trace info." of Table 2.3. Reasons for missing traces include: the method for which we tried to generate a trace is a native method, the Android CTS lacks tests for the given method, or an error occurred while instrumenting the class under test or while running the tests. This last case often took the form of triggering internal responsiveness timers inside the Android OS, known as ANR (Application Not Responding) [49]—because our instrumentation results in a significant slowdown (about 20x), these timers are triggered more often than they would be in uninstrumented runs. Since capturing the traces is a one-time activity, this high overhead is otherwise acceptable.

These results suggest that Modelgen can be used to replace most of the effort involved in constructing manual models, since it reproduced almost all our manual flow annotations (96.38% recall) and produced many new correct annotations that our existing models lacked. Although our evaluation focuses on Java and Android, the results should generalize to any platform for which good test suites exist.

2.5.2 Whole-System Evaluation of STAMP and Modelgen

The STAMP static analysis component is a bounded context-sensitive, flow- and path-insensitive information flow analysis. A complete description of this system can be found in Section 4 of [38]. STAMP never analyzes platform code and treats platform methods for which it has no explicit model under best-case assumptions. That is, platform methods without models are assumed to induce no flows between their arguments or their return values. The alternative, analyzing under worst-case

assumptions, produces an overwhelming number of false positives.

To evaluate the usefulness of our specifications in a full static analysis, we ran STAMP under two configurations: base and augmented. In the base configuration, we used only the existing manually-written models. In the augmented configuration, we included (1) all source and sink annotations from the manual models (annotating sources and sinks is outside of the scope of Modelgen), (2) the Modelgen specifications generated in the experiment of Section 2.5.1, and (3) the existing manual models for those methods for which Modelgen did not construct any specifications (e.g. the `java.lang.*` classes). The base and augmented configurations included 1215 and 2274 flow annotations, respectively.

We compared the results of both configurations on 242 apps from the Google Play Store. These apps were randomly selected among those for which STAMP was able to run with a budget of 8GB of RAM and 1 hour time limit in both configurations. The average running time per app is around 7 minutes in either configuration.

STAMP finds a total of 746 (average 3.08 per app) and 986 (average 4.07) flows in the base and augmented configuration, respectively. The union of the flows discovered in both configurations is exactly 1000. In other words, STAMP finds 31% (254) new flows in the augmented configuration. Like most static analysis systems, STAMP can produce false positives, even when given sound models. Additionally, Modelgen may produce unsound models for some methods (recall the discussions in sections 2.4.4 and 2.4.5). Given this, we would like to know what proportion of these new flows are true positives. To estimate the true positive rate of the new flows, we took 10 random apps from the subset of our sample (109 of 242 apps) for which the augmented configuration finds any new flows. We manually inspected these apps and marked those flows for which we could find a feasible source-sink path, and for which control flow could reach such path, as true positives. Although this sort of inspection is always susceptible to human error, we tried to be conservative in declaring flows to be true positives. In most cases, the flows are contained in advertisement libraries and would trigger as soon as the app started or a particular view within the app was displayed to the user.

Figure 2.10 shows the results of our manual inspection. The flows labeled as "Augmented configuration: Unknown" are those for which we could not find a source-sink path, but are not necessarily false positives. The flows labeled "Augmented configuration: True Positives" represent a lower bound on the number of new true positives that STAMP finds only in the augmented configuration. The lower portion of the bar corresponds to those flows found in both configurations, without attempting to distinguish whether they are false or true positives. For the 10 apps, the augmented configuration produces 64% more flows than the base configuration, and at least 55% of these new flows are true positives.

The recall of the augmented configuration, which is the percentage of all flows found in the base configuration that were also found in the augmented configuration, is 98.12%. A flow found in the base configuration could be missed in the augmented configuration if Modelgen infers a different

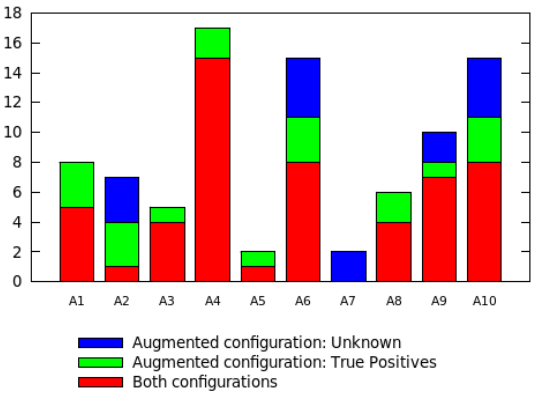


Figure 2.10: Flows found with and without Modelgen models

specification for a method, which is relevant for the flow, than the manually-written model.

2.5.3 Controlling for Test Suite Quality

Specification mining based on concrete execution traces depends on having a representative set of tests for each method for which we want to infer specifications. One threat to the validity of our experiment is that it could be that our results are good only because the Android compatibility tests are unusually thorough. In this section we attempt to control for the quality of the test suite.

We measure how strongly our specification mining technique depends on the available tests by the number of method executions it needs to observe before it converges to the final specification. Intuitively, if few executions of a method are needed to converge to a suitable specification of the method's behavior, then our specification mining technique is more robust that if it requires many executions, and therefore many tests. Additionally, if a random small subset of the observed executions is enough for our technique to discover the same specification as the full set of executions, we can gain some confidence that observing additional executions won't dramatically alter the results of our specification mining.

We take all methods from Table 2.3 for which we are able to record traces and Modelgen produces non-empty specifications, which are 264 methods in total. We restrict ourselves to those methods, as opposed to the full set for which we have mined specifications, since we have examined them and found them to be correct during the comparison of Section 2.5.1. For each such method m, we consider the final specification produced by Modelgen (S_m) as well as the set \$ of specifications for each invocation subtrace of m. Starting with the empty specification we repeatedly add a random specification chosen from \$ until the model matches S_m, recording how many such subtrace specifications are used to recover S_m.

Figure 2.11 shows a log scale plot of the number of methods (vertical axis) that required n traces (horizontal axis) to recover the full specification over each of 20 trials. That is, we sampled the

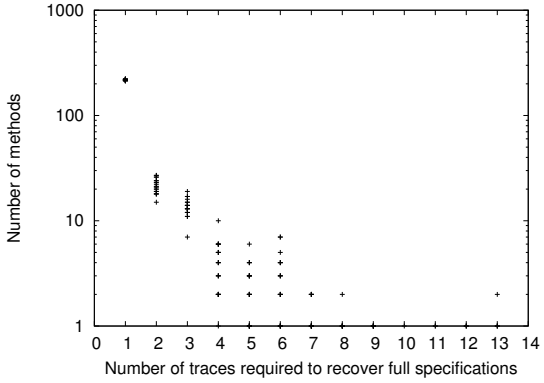


Figure 2.11: Specification Convergence

executions of each method to recover its specification and then counted the number of methods that needed one execution, the number that needed two, and so on, and then repeated this process 19 more times. The multiple points plotted for each number of executions give an idea of the variance due to the random choices of method executions to include in the specification.

It is also useful to consider aggregate statistics over all method specification inferences. In our experiment, 83.7% of the methods needed just one subtrace specification to recover the specification and no method required more than an average of 9 random subtrace specifications. The maximum number of subtraces needed to converge to a method specification (when taking the worst out of 20 iterations of the algorithm) was 13 for `java.util.Vector.setElementAt(Object, int)`. The average number of subtraces required to converge to a specification is 1.38. For comparison, the specifications evaluated in Section 2.5.1 were inferred using a median of 4 traces (the average, 207, is dominated by a few large outliers). We conclude that explicit information flow typically requires few observations to produce useful specifications.

Chapter 3

Minimizing GUI Event Traces

Recall from the previous chapter that dynamic analysis is effective only when it is possible to trigger representative executions that explore the relevant behavior of the program being analyzed. In the case of our specification mining technique, we achieved this exploration of the Android platform methods, by leveraging an existing comprehensive test suite. However, in general, obtaining a set of executions thoroughly sampling the behavior of a program we care about is non-trivial. This is particularly the case when we wish to sample the behavior of application code, or of a specific application's interaction with the platform code, since we rarely have access to any reasonable test suite for third-party applications.

While we could write such a test suite ourselves, even for applications for which we are missing the source code, this is a time consuming process and essentially amounts to indirectly writing an implicit specification of the relevant application behavior in the form of our test suite. Ideally, we would like to have a method to generate executions sampling relevant behavior for a given application, which we can then use to either generate reproducible tests or to produce executions which can directly drive a dynamic analysis system.

An additional benefit of any automatic exploration system is that it also helps first-party developers produce useful test suites for their applications. Test cases are time consuming to write, especially for applications dealing with rich graphical user interfaces (GUIs). Many properties can only be reliably tested once the program has reached a particular state, such as a specific screen or view in its GUI. Part of the challenge of GUI testing is in creating a sequence of user interactions that cause the program to reliably reach a target GUI state, under which the test one cares about can be performed.

Automatically generating a sequence of GUI events to reach a particular point in the program is a difficult problem to solve in general. In many cases, it is possible to reach the desired point in an application by randomly generating GUI events until the right view is displayed, or by capturing and replaying the GUI events generated by a user, or a human tester, interacting with the application.

However, these randomly generated or tester-captured *GUI event traces* generally contain more interactions than necessary to reach the desired state. Furthermore, traces generated by capture and replay of concrete user interactions might not be robust in the face of application non-determinism, and thus might break if the program changes behavior, even slightly, between executions. In modern mobile and web apps, which often include internal A/B testing logic and personalization, GUI non-determinism is a common obstacle for simple capture and replay systems.

In this chapter, we:

- Present an algorithm based on delta-debugging [117, 119] to minimize a trace of GUI events. Our algorithm is robust to application non-determinism. When the application is deterministic, the output trace is 1-minimal, meaning no element of the trace can be removed without changing the behavior we care about [119].
- Minimization proceeds by checking whether subtraces of a trace still reach the desired state. This problem is highly parallelizable, but it is not obvious how to take maximal advantage of the information gained from each subtrace trial (see Section 3.3). We define the subtrace selection problem and provide an efficient heuristic, as well as an optimal solution based on the problem's characterization as a Markov Decision Process (MDP).
- We show the effectiveness of the above techniques in minimizing randomly generated traces for two representative datasets: one set of popular Android apps taken from the Google Play Store, and a set of open-source apps from the F-droid application repository.

One advantage of our trace minimization technique is that it treats the application in a blackbox manner, and thus is robust to implementation details such as the programming language, UI toolkit or server/client distribution of the application logic. Our approach should be valid for any Android app which can be tested using Monkey [50], a tool that generates random sequences of GUI events, avoiding the relative fragility of research tools that rely on the application, for example, exposing higher-level accessibility information or being amenable to whitebox analysis techniques.

Section 3.1 provides a brief overview of the problem of GUI trace minimization in the face of application non-determinism, the relevant characteristics of the Android platform, and a motivating example for our technique. Section 3.2 describes our trace minimization algorithm while Section 3.3 explores the problem of subtrace selection embedded in the algorithm's inner loop, including the characterization as a Markov Decision Process (3.3.3). Section 3.4 presents our results, contrasting the performance of different solutions to the subtrace selection problem.

3.1 Problem Overview

The GUI (Graphical User Interface) of an Android application consists of a set of *activities*. Each activity corresponds to a top-level view (e.g., a page or a screen) of the user interface. Additionally,

each app has one *main activity*, which is the first one invoked when the app is launched. In the course of interacting with an application's GUI, the user commonly transitions between different activities, as actions in one activity trigger another activity.

A *GUI event trace* is a sequence of user interface events such as screen taps, physical keyboard key presses and complete multi-touch gestures. One way to obtain such a trace is to record the actions of a user. Another option uses a random or programmatic input generation tool to generate the trace without human involvement. In our experiments, we took the second approach, using Monkey, a standard first-party random input generation tool for Android.

The standard way of running Monkey involves having it generate random events live on top of a running app. However, we use an alternate mode, in which the full event trace is generated once in the beginning, and then replayed any number of times on top of a target app. The input to Monkey includes the number and distribution (i.e. the proportion for each type, such as taps versus keyboard key presses) of desired events. Given such input, Monkey will generate a trace of random events, which can then be replayed for a particular app. A GUI event trace is replayed by Monkey sending each event in turn to the selected app, which runs on a connected device or emulator, and pausing a configurable time interval between one event and the next.

By repeating this process and generating multiple large random traces for each app under test, we are able to reach various activities within the app. These traces could conceivably serve as system tests for the app, especially if augmented by state checks or log monitoring. However, using automatically generated traces as tests can be problematic for the following reasons:

1. Because traces are randomly generated, the majority of the events do not trigger any interesting app behavior (e.g. they are taps on inactive GUI elements), or trigger behavior unrelated to the functionality we care about for a given test. Large random GUI event traces are hard for humans to interpret, particularly if most of the trace is irrelevant for the desired behavior. The traces produced by Monkey typically consist mostly of irrelevant events.

2. Replaying a large trace is a time consuming process, as a delay must be introduced before sending each event to the app, to ensure previous events have been fully processed. In our experiments, we set this delay to 4 seconds, which we found necessary to allow for events that trigger network requests or other expensive operations to be fully handled by the app before the next event is triggered. This threshold could be adjusted dynamically, but possibly not without a more invasive (white-box) instrumentation of the running app's code or the Android platform itself.

Due to the above issues, given a large event trace, we would like to find a minimal subtrace that triggers the same app behavior. In particular, we focus on subtraces that reach the same activity. Specifically, if we assume that the app is deterministic in the sense that the same event trace always visits the same activities in the same order, then the algorithm we shall give in Section 3.2 extracts,

from a given event trace that reaches a particular activity, a 1-minimal subtrace that reaches that activity. As we will show later in this section, Android applications often exhibit non-deterministic behavior, and we will also formulate a notion of approximate 1-minimality that generalizes to non-deterministic GUI behavior.

Note that we use activity reachability as a proxy for uncovering user triggered behavior in the app. All the techniques and checks in this chapter apply just as well if the behavior we seek to trigger involves reaching the execution point of a particular GUI widget callback or the point at which a particular web request is sent. We only require that we have a small finite set of behaviors that shall be triggered in the app, so that every system test is built on top of a minimal trace triggering that behavior. In the rest of this chapter, we assume that we seek minimal GUI event traces that cause a particular activity to be reached, but extending it to other notions of reachable behavior is straightforward, so long as we have a way to test if the behavior is triggered or not, when running a particular event trace.

3.1.1 Trace Minimization Example and Application Non-determinism

One of the apps we use for our experiments (see Section 3.4) is Yelp's Eat24 (`com.eat24.app.apk`), a popular commercial food delivery app. To generate minimized traces that reach this app's activities, we run the app on an Android emulator and use Monkey to generate multiple GUI event traces. Each trace consists of 500 single-touch events at random coordinates within the app's portion of the screen. To capture non-determinism in the app, we replay each trace multiple times (we use 20 repetitions), clearing any local data or app state in the device between replays.

For a particular trace T we generated, the activity `LoginActivity` is always reached by replaying T on this app. `LoginActivity` is a relatively easy to reach activity for this particular app, as there are at least two ways to launch this activity immediately from the app's main activity: either by clicking on an item in the application's menu or on the lower portion of the app's home screen (which displays recommendations if the user is already logged in). The second method requires only a single GUI event: the click on the bottom of the screen. However, approximately 50% of the time when the app is launched from a clean install, it shows a dialog asking for the user's address and zip code. This dialog blocks interaction with the main activity and, in our set-up, automatically brings up Android's software keyboard. Dismissing the dialog is as simple as clicking anywhere in the screen outside the dialog and the virtual keyboard. However, this does not dismiss the keyboard itself, so the state of the screen is different than if the dialog had never appeared. After dismissing the dialog, it is still possible to navigate to the `LoginActivity` with one more click, but the area of the screen that must be clicked is different than if the dialog had never appeared at all.

Suppose now we wanted to manually select, out of the 500 events T, a minimal subtrace T' such that replaying the events of T' in order reaches the `LoginActivity` activity regardless of whether the app shows the location dialog or not. This subtrace must exist, since the original trace always

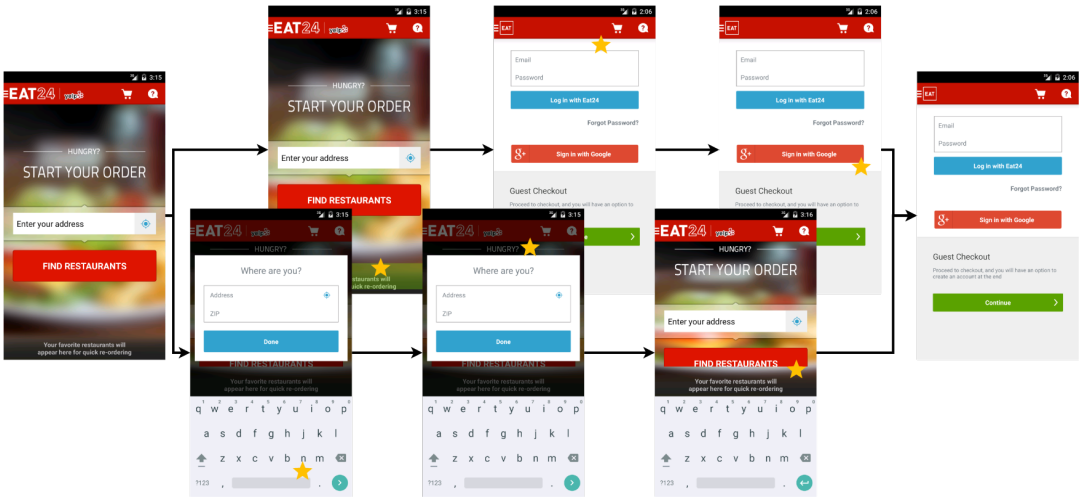


Figure 3.1: Reaching `LoginActivity` on `com.eat24.app.apk`

reaches `LoginActivity` (among other activities), in both cases. However, we cannot simply select the single click subtrace that would launch `LoginActivity` if the dialog is not present, since it won't work when the dialog does appear. We have the same problem if we focus on picking the two clicks when the dialog appears, as such a trace does not necessarily work when the dialog is absent. For example, most clicks that would simply dismiss the dialog might also trigger different behavior if the dialog is missing by clicking on active GUI widgets underneath the dialog.

This sort of behavior is not exclusive to the EAT24 app. In fact, many Android apps behave non-deterministically, either because of internal A/B testing logic or just because of the non-determinism of the app's external environment. We could always manually write GUI testing scripts that are aware of the app's specific non-determinism and generate different GUI events when faced with different app states. However, as we will show, it is possible to automatically generate small subtraces that are robust against application non-determinism, while still treating the app itself as a black box. We require only a way to run subtraces on top of the app from an initial state multiple times and list the activities being reached.

Figure 3.1 shows the execution of a 3 event trace – the minimal subtrace obtained by the technique in this chapter – that accomplishes our goal without checking at any point the state of the application's GUI. If there is no location dialog, the first event in the trace triggers the direct transition to `LoginActivity` by clicking on the bottom of the screen. The second and third events click inactive areas of the `LoginActivity` GUI layout, having no effect in this case. If the dialog appears, the first click hits a portion of the virtual keyboard layout, typing a whitespace character into the location dialog. The second click immediately dismisses the dialog without using the whitespace character. Finally, the third click happens in the new location of the panel that launches

`LoginActivity`, without dismissing the keyboard[1]. Thus, whether or not the dialog appears, the script will reach `LoginActivity` and stop there. It is worth noting that at no point is our technique aware of the dialog; it only knows that this script reaches `LoginActivity` with high probability over multiple runs of the app.

3.2 Minimization Algorithm

In this section we present our trace minimization algorithm and discuss its basic properties. Our algorithm is based on Zeller's delta debugging algorithm (see [119]), reformulated for our problem and augmented to deal with application non-determinism.

Intuitively, delta debugging starts with a large input sequence that passes a particular test oracle and attempts to remove part of the sequence at every step, such that the remaining subsequence still passes the oracle. This is repeated until we have a 1-minimal subsequence that is accepted by the oracle. In general, a *1-minimal* subsequence with property P is one which satisfies P, but where no subsequence which can be obtained from it by removing any single element satisfies P.

Our version of delta debugging takes as input an Android app A, a trace T and a target activity a within A. A trace is an ordered sequence $T = \{e_0, e_1, ...\}$ of GUI events. A subtrace T' of T is a subsequence of T. The algorithm has access to a test oracle $O(A, T, a)$, which consists of starting a new Android emulator, installing A, running trace T on A and verifying that a was launched during that execution. The oracle returns either 0 (the target activity a was not reached) or 1 (activity a was reached). Because of application non-determinism, our test oracle may accept a trace with some probability p, and reject it with probability $1 - p$. Fixing A and a, we define the probability $P_T = Pr[O(A, T, a) = 1]$, which is the trace's underlying probability of reaching activity a when run on app A.

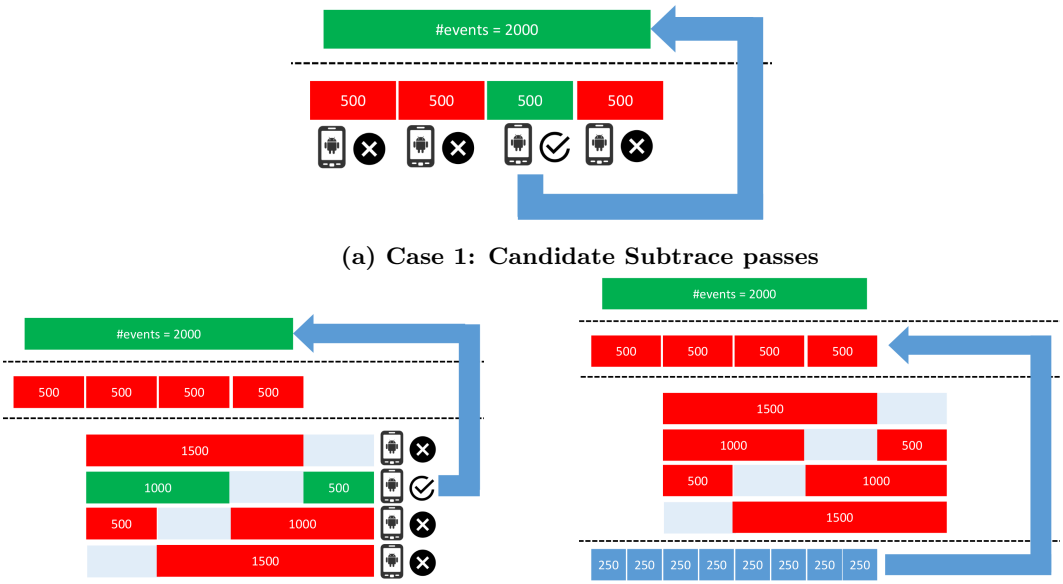
Figure 3.2 shows the pseudo-code for the general form of our trace minimization algorithm (`ND3MIN`). Besides A, T and a, the algorithm uses 4 global parameters: the oracle O, a starting number n of subtrace candidates to test, a number of times to run each candidate (nr) and the success threshold required to accept it (st).

For the algorithm to select a particular subtrace T' at the end of any minimization step, calling $O(A, T', a)$ nr times results in the oracle returning 1 at least st times. This requirement is enforced by function `passes()` in line 20. We say that a trace is *successful* if it passes the check in `passes()`. Function `get_passing()` in line 16 takes a set S of subtraces and selects a subtrace $T_c \in S$ such that `passes(`A`,`T_c`,`a`)` returns true. It returns None iff no such subtrace exists in set S. Note that `get_passing()` specifies no order in which traces are passed to `passes()`. We assume oracle calls are expensive but we have the ability to make multiple oracle calls in parallel. In Section 3.3, we use the flexibility in the definition of `get_passing()` to minimize the number of rounds of (parallel)

[1] Transitioning between activities does dismiss the keyboard, so we don't need to do so explicitly.

```
     globals : O, n, nr, st
 1   def ND3MIN(A,T,a):
 2       return MinR(A,T,a,n);
 3   def MinR(A,T,a,k):
 4       size ← len(T)/k;
 5       for i in range(0,k):
 6           T_i ← T[i * size : (i + 1) * size]; T̄_i ← T \ T_i;
 7       T_sub ← get_passing(A,{∀i ∈ [0,k). T_i},a);
 8       if T_sub ≠ None:
 9           return MinR(A,T_sub,a,n);
10       T_compl ← get_passing(A,{∀i ∈ [0,k). T̄_i},a);
11       if T_compl ≠ None:
12           return MinR(A,T_compl,a,max(k − 1,2));
13       if k < len(T):
14           return MinR(A,T,a,min(2k,len(T)));
15       return T;
16   def get_passing(A,S,a):
17       if ∃T_c ∈ S. passes (A,T_c,a):
18           return T_c;
19       return None;
20   def passes(A,T,a):
21       s ← 0;
22       for i in range(0, nr):
23           s ← s + O (A,T,a);
24       return s ≥ st;
```

Figure 3.2: ND3MIN: Non-Deterministic Delta Debugging MINimization.



(a) Case 1: Candidate Subtrace passes

(b) Case 2: Complement Subtrace passes

(c) Case 3: Increase k

Figure 3.3: Delta Debugging: Cases

calls to O required by our algorithm.

Conceptually, we wish to mantain the invariant that the subtrace T' at the end of any minimization step succeeds st or nr times, in order to produce a reliable test trace at the end of the process, which needs only be run a few times to reach the desired behavior. This requirement is sometimes in opposition to that of producing a small minimized subtrace, and the values of st and nr regulate this trade-off. As we see later in this chapter, even for relatively high success thresholds, the resulting minimized traces are often quite small.

`ND3MIN()` calls the recursive function `MinR()` which uses the two helper functions described above to implement our version of delta debugging. `MinR()` follows the classic structure of delta debugging. First (lines 4–6), it partitions trace T in k subtraces T_i of contiguous events of roughly equal size (starting with $k = n$ on the first recursive call). It also generates the complement of each subtrace T_i, defined as $\overline{T_i} \leftarrow T \setminus T_i$. It then proceeds in three alternating cases (depicted in Figure 3.3) and a fourth terminal case:

Case #1: If any candidate subtrace T_i is successful, according to `get_passing()`, the algorithm selects that T_i as its new current trace T, and calls itself recursively with $k = n$ (lines 7–9, Figure 3.3a).

Case #2: Otherwise, if any complement subtrace $\overline{T_i}$ is successful, the algorithm selects that $\overline{T_i}$ to be the next T. If $k \geq 3$, $k = k - 1$ before the next recursive call, otherwise it is set to 2 (lines 10–12, Figure 3.3b). Reducing k by 1 ensures that the candidate subtraces T_i generated

in the next call will be a subset of those in this call, so the algorithm will keep reducing to complement subtraces until $k = 2$ or no $\overline{T_i}$ is successful. Note that when $k = 2$, both the set of candidate subtraces and that of complements are identical ($\overline{T_0} = T_1$ and $\overline{T_1} = T_0$).

Case #3: If k is smaller than the number of events left in T, we double the number of partitions, up to $k = \text{len}(T)$ (lines 13–14, Figure 3.3c). Note that when we reach $k = \text{len}(T)$, this implies that on the next recursive call, every subtrace consists of a single event.

Case #4: Otherwise, return T as our minimized subtrace T_{min} (line 15).

Note that Case #4 happens only when the number of partitions equals the size of the current trace T, meaning each candidate subtrace is of size 1 (i.e. contains a single event). Additionally, it can only be reached when no complement subtrace succeeds. In the deterministic case, this makes T_{min} 1-minimal by definition: since we have observed it to succeed according to `get_passing()`, it must be the case that it reaches the desired activity a when replayed on the app A and, at the same time, we have tested every subtrace obtainable from T_{min} by removing a single element (the complement subtraces) and found that it fails to reach a. On our non-deterministic setting, however, things are slightly more complicated. In the next subsection we will briefly discuss what we can and cannot say about `ND3MIN` in the cases where the app behaves non-deterministically.

3.2.1 Non-determinism and Properties of our Algorithm

Ideally, we would like to show that, given a probability (lower) bound p_b, if $P_T \geq p_b$ and $T_{min} =$ `ND3MIN(`A, T, a`)` then, with high probability, T_{min} is a 1-minimal subtrace of T such that $P_{T_{min}} \geq p_b$. Unfortunately, this is not possible. To see why, imagine $\exists T' \subsetneq T_{min}$, $P_{T'} = p_b - \delta$. As $\delta \to 0$, the number of checks required to distinguish T' from a subtrace which fulfills $P_{T'} \geq p_b$ grows without bound. If T' can be obtained from T_{min} by removing a single event, then testing T_{min} for 1-minimality must take unbounded time. We consider instead the following property:

Definition 1. *A trace T is* approximately 1-minimal *with respect to the bound p_b and distance ϵ, if $P_T \geq p_b - \epsilon$, and any subtrace T' which can be obtained by removing a single event from T is such that $P_{T'} < p_b$.*

We would like to bound the probability that `ND3MIN` returns an approximately 1-minimal trace with bound $p_b = st/nr$ and distance ϵ. We would first like to show that if $P_T \geq p_b$ and $T_{min} =$ `ND3MIN(`A, T, a`)`, then, with high probability, $P_{T_{min}} \geq p_b - \epsilon$ for a small ϵ. Given a single independent subtrace T', if the check $X =$ `passes(`A, T', a`)` returns true, then the probability that $P_{T'} \geq p_x$, for a given p_x, is:

$$\tilde{p} = Pr[P_{T'} \geq p_x \mid X] \tag{3.1}$$

$$= \int_0^1 Pr[P_{T'} \geq p_x \mid X, P_{T'} = p] \cdot f_{P_{T'}|X}(p)dp \tag{3.2}$$

$$= \int_{p_x}^1 f_{P_{T'}|X}(p)dp \tag{3.3}$$

where $f_{P_{T'}|X}(p)$ is the probability density of $P_{T'}$ given X. Note that $Pr[P_{T'} \geq p_x \mid P_{T'} = p]$ is 1 if $p \geq p_x$, and 0 otherwise. By Bayes' theorem:

$$\tilde{p} = \frac{\int_{p_x}^1 Pr[X \mid P_{T'} = p] \cdot f_{P_{T'}}(p)dp}{Pr[X]} \tag{3.4}$$

$$= \frac{\int_{p_x}^1 Pr[X \mid P_{T'} = p] \cdot f_{P_{T'}}(p)dp}{\int_0^1 Pr[X \mid P_{T'} = p] \cdot f_{P_{T'}}(p)dp} \tag{3.5}$$

By definition of $X = \texttt{passes}(A, T', a)$:

$$Pr[X \mid P_{T'} = p] = \sum_{i=st}^{nr} \binom{nr}{i} \cdot p^i (1-p)^{nr-i} \tag{3.6}$$

Note that if we assume a discrete probability distribution, we can replace the integrals above by sums over p, and the density functions $f_{P_{T'}}(p)$ by probabilities $Pr[P_{T'} = p]$. Using this approximation and plugging the parameters $nr = 20$ and $st = 18$, as well as the (discrete) prior probability distribution $Pr[P_{T'} = p]$ obtained experimentally (see Section 3.4), we have $Pr[P_{T'} \geq 0.85 \mid \texttt{passes}(A, T', a)] > 0.95$. So, selecting $p_b = 0.9$ and $\epsilon = 0.05$ would at first seem like an option to prove a bound on the probability of $P_{T_{min}} \geq p_b - \epsilon$. Unfortunately, most executions of our algorithm perform a large number of calls to `passes()`, and the error accumulates rapidly. After just 20 calls, the naive bound on $Pr[P_{T_{min}} \geq 0.85]$ in our example would become $0.95^{20} \approx 0.36$. Bounding the error in our algorithm more tightly is non-trivial. Instead, when running our experiments, we perform a final independent check, calling $\texttt{passes}(A, T_{min}, a)$ one last time on the final output of our algorithm. In Section 3.4 we observe that this final check passes often. For the minimized traces where this final check succeeds, we can indeed say that $P_{T_{min}} \geq 0.85$ with probability > 0.95, as per the example above, which uses our experimental parameters.

We can now get a bound on the probability of the second requirement in the definition of approximate 1-minimality:

Lemma 1. *If $T_{min} = \texttt{ND3MIN}(A, T, a)$, then the probability $\widehat{p}$ that there exists no subtrace T',*

obtained by removing a single event from T_{min}, such that $P_{T'} \geq p_b$ is:

$$\widehat{p} = \left(1 - \frac{\int_{p_b}^{1} \left(1 - \sum_{i=\mathrm{st}}^{\mathrm{nr}} \binom{\mathrm{nr}}{i} \cdot p^i (1-p)^{\mathrm{nr}-i}\right) \cdot f_{P_{T'}}(p) dp}{\int_{0}^{1} \left(1 - \sum_{i=\mathrm{st}}^{\mathrm{nr}} \binom{\mathrm{nr}}{i} \cdot p^i (1-p)^{\mathrm{nr}-i}\right) \cdot f_{P_{T'}}(p) dp}\right)^l$$

where l is the number of events in T_{min}.

Proof. Consider only the last call to the recursive procedure `MinR()`, which returns T_{min}. Note that `MinR()` returns T_{min} only in Case #4 which executes only when cases #1 to #3 are not satisfied. Thus, for the last execution of `MinR()`, Case #3 must have been skipped, which means $k \geq \text{len}(T_{min})$. Since $k \geq \text{len}(T_{min})$, the set $\{\forall i \in [0, k).\, \overline{T_i}\}$ contains every subtrace which can be obtained by removing a single event from T_{min}. Because Case #2 was also not satisfied, we know that calling `get_passing()` on this set at line 10 returns **None**. By definition, this is equivalent to calling `passes()` on each $\overline{T_i}$ and having it return false.

Taking $Y_i = \neg$`passes(`A,$\overline{T_i}$,a`)`, we have:

$$Pr[Y_i \mid P_{\overline{T_i}} = p] = 1 - \sum_{i=st}^{nr} \binom{nr}{i} \cdot p^i (1-p)^{nr-i} \tag{3.7}$$

And, using steps analogous to equations 3.1 to 3.5 above:

$$\widehat{p_i} = Pr[P_{\overline{T_i}} \geq p_b \mid Y_i] \tag{3.8}$$

$$= \frac{\int_{p_b}^{1} Pr[Y_i \mid P_{\overline{T_i}} = p] \cdot f_{P_{\overline{T_i}}}(p) dp}{\int_{0}^{1} Pr[Y_i \mid P_{\overline{T_i}} = p] \cdot f_{P_{\overline{T_i}}}(p) dp} \tag{3.9}$$

which is the probability $Pr[P_{\overline{T_i}} \geq p_b]$ for each $\overline{T_i}$ given the behavior of `passes()` the algorithm must have observed.

Finally, the probability that no $\overline{T_i}$ is such that $P_{\overline{T_i}} \geq p_b$ is given by $\widehat{p} = \prod_{\overline{T_i}} (1 - \widehat{p_i})$ which expands to the formula for $\widehat{p}$ given in the lemma's statement, since the formula for $\widehat{p_i}$ is the same for every T_i, and there are $l = k = \text{len}(T_{min})$ such subtraces. □

3.3 Trace Selection

Recall that the method `get_passing()` takes a set S of subtraces of T, and must return a subtrace $T_c \in S$ such that calling the oracle $O\,(A, T_c, a)$ nr times would succeed st times. If no such $T_c \in S$ exists, `get_passing()` must be able to determine that and return **None**. Calls to oracle O are time consuming, so we wish to minimize the number of such calls that execute non-concurrently.

We assume that we have a maximum of m instances of O which can be run in parallel. In our implementation, these represent individual instances of the Android emulator.

For each call to `get_passing()`, we have a set of n traces $S = \{T_0, ..., T_{n-1}\}$. We define a schedule as an array $sch = [v_0, ..., v_{n-1}]$ such that $\sum_{j=0}^{n-1} v_j \leq m$. A step for `get_passing()` consists on generating a new schedule sch, running v_j oracle calls $O(A, T_j, a)$ for each $j \in [0, n-1]$ and capturing the results. Since the total number of calls to O is at most m, the calls corresponding to a single step of `get_passing()` can be executed in parallel. We accumulate the results of all steps before the current one as pairs (s_j, f_j), where s_j is the number of successes seen so far for T_j (i.e. 1 was returned by $O(A, T_j, a)$) and f_j the number of failures (0 was returned). We can see that given the definition of `get_passing()`, we never gain anything from running a single T_j more than nr times, so we forbid this, and thus $\forall j.\, s_j + f_j \leq nr$. We seek to minimize the number of steps in each call to `get_passing()` before we can either identify a T_j which satisfies `passes()` (i.e. $\exists j.\, s_j \geq st$) or we have concluded that no such T_j can exist. We note that, given the previous constraints, if ever $f_j > nr - st$, T_j cannot be a trace that satisfies `passes()`.

We give 3 strategies for minimizing the number of steps of `get_passing()`. Section 3.4 compares them empirically.

3.3.1 Naive Scheduling

There are two obvious ways to generate the schedule sch at every step of `get_passing()`, which depend very little on the observed pairs (s_j, f_j).

The first method is to schedule all nr executions for each trace one after the other, so at every step $v_j = min(nr - s_j - f_j, m - \sum_{k=0}^{k<j} v_k)$. This strategy goes from $j = 0$ to $n-1$ and greedily tries to add another execution of T_j to sch until doing so would either mean that more than nr executions of T_j have been scheduled over all steps of `get_passing()` or would push the schedule beyond the limit of m calls to O. A common sense optimization is, at every step, to return immediately if a T_j with $s_j \geq st$ has been found, and ignore any T_j with $f_j > nr - st$ for scheduling. The worst-case for this strategy happens when no trace in S satisfies `passes()`. The following is a particular example of this greedy strategy in action with $nr = 20$, $st = 18$, $n = 3$ and $m = 15$. We represent each step as a transition between two lists of pairs (s_j, f_j) $\forall j = 1, 2, 3$, representing the accumulated results before and after the step. Each step is also annotated with the corresponding schedule sch:

$$[(0,0),(0,0),(0,0)] \xrightarrow[15,0,0]{} [(13,2),(0,0),(0,0)]$$

$$[(13,2),(0,0),(0,0)] \xrightarrow[5,10,0]{} [(17,3),(3,7),(0,0)]$$

$$[(17,3),(3,7),(0,0)] \xrightarrow[0,0,15]{} [(17,3),(3,7),(15,0)]$$

$$[(17,3),(3,7),(15,0)] \xrightarrow[0,0,5]{} [(17,3),(3,7),(19,1)]$$

Another naive strategy is to schedule traces in a round–robin fashion. Each step scans $j = 0$ to

$n-1$ multiple times, adding an additional invocation of T_j if $s_j + f_j + v_j \leq nr$ and $f_j \leq nr - st$. It stops when the schedule is full (m calls scheduled) and proceeds to run *sch*. Again, we stop as soon as there is nothing else to run or we have found a T_j with $s_j \geq st$. The worst-case for this strategy happens when all traces succeed with high probability. We repeat the example above with the round–robin strategy:

$$[(0,0),(0,0),(0,0)] \xrightarrow[5,5,5]{} [(4,1),(2,3),(5,0)]$$
$$[(4,1),(3,2),(5,0)] \xrightarrow[8,0,7]{} [(11,2),(2,3),(12,0)]$$
$$[(11,2),(2,3),(12,0)] \xrightarrow[7,0,8]{} [(17,3),(2,3),(19,1)]$$

Both of these naive strategies are similar in that they are likely to do plenty of unnecessary work in the average case. We chose the round–robin variant as our baseline for comparison, since it performs better in the case in which is common for many of the traces in S to fail the oracle often and st is close to nr (this matches our scenario).

3.3.2 Heuristic: Exploration Followed by Greedy

The naive algorithms don't exploit all of the information contained in the pairs (s_j, f_j) in deciding what to do next. Clearly, if we have a trace T_{j_1} for which $(s_{j_1}, f_{j_1}) = (5, 0)$, and a trace T_{j_2} for which $(s_{j_2}, f_{j_2}) = (3, 2)$, then T_{j_1} is significantly more promising than T_{j_2}, and we should try checking it first. Conversely, it should be possible to discard T_{j_2} by scheduling it for execution only a few more times; adding 15 copies of T_{j_2} to the next schedule, while allowed, would likely be a waste. We use these observations to propose a heuristic that, at every step: a) tries to confirm traces that seem likely to be successful, and b) tries to discard traces that seem likely to be unsuccessful, in that order of priority.

Before scheduling the first step, we have $\forall j.\,(s_j, f_j) = (0, 0)$. Since we have no information, we simply schedule the traces in S in a round–robin fashion. This gives us at least some information about every T_j. In every round after that, we follow the algorithm outlined in Figure 3.4.

We first compute $p_j = s_j/(s_j + f_j)$ for each j, the empirical success probability of T_j so far. We sort S in descending order, first by p_j and then by s_j. Then we partition the sorted array S into two sections: S_c contains the traces such that $p_j \geq c$ for a certain constant c ($c = 0.8$ in our implementation) and S_f the rest. We assume that traces in S_c are likely to succeed after nr calls to O, while traces in S_f are likely to fail, and we predict accordingly. While there are traces in S_c, and our schedule is not full, we remove T_k from S_c in order. We compute x_k in line 11, which is the expected number of runs of T_k needed to get to the point where $s_k = st$. If we can schedule that many runs, we do so (line 13). If we can't schedule x_k runs of T_k in this step, but we can do it in the next step, we move T_k to S_d, a list of 'deferred' traces from S_c. We do this so that if we can pack the expected number of runs for multiple traces in S_c we do so, even if those aren't the traces

```
     globals : c
 1   def schedule(S,[(s_j, f_j)]):
 2       ∀j. v_j ← 0;
 3       ∀j. p_j ← s_j/(s_j + f_j);
 4       sort S by p_j, s_j desc;
 5       S_c ← [T_j ∈ S | p_j ≥ c];
 6       S_d ← ∅;
 7       S_f ← [T_j ∈ S | p_j < c];
 8       while Σ_j v_j < m:
 9           if S_c ≠ ∅:
10               T_k ← remove_first(S_c);
11               x_k ← min(nr − s_k − f_k, ⌈(st − s_k)/p_k)⌉);
12               if x_k ≤ m − Σ_j v_j:
13                   v_k ← x_k;
14               elif x_k ≤ m:
15                   S_d ← S_d ∪ [T_k];
16               else:
17                   v_k ← m − Σ_j v_j;
18           elif S_d ≠ ∅:
19               Schedule from S_d by round–robin.;
20           elif S_f ≠ ∅:
21               T_k ← remove_first(S_f));
22               y_k ← min(nr − s_k − f_k, ⌈ (nr−st+1)−f_k / (1−p_k) ⌉);
23               v_k → min(y_k, m − Σ v_j);
24           else:
25               Schedule from original S by round–robin.;
```

Figure 3.4: Trace Selection Heuristic.

with the highest p_j. If x_k is too large to schedule in any single step, then we just schedule as many copies of T_k as we can.

Only when S_c is empty, we revisit the traces in S_d and schedule them in round–robin fashion. If there are no traces in S_d then we begin removing T_k from S_f in order. We compute y_k in line 22 for these traces, which is the expected number of runs of T_k needed to get to the point where $f_k = nr - st + 1$ (at which time we can declare that trace as failing `passes()`). We could run the traces in S_d in order of increasing empirical success probability, which would allow us to discard some of them more quickly, but this doesn't reduce the expected number of steps for `get_passing()`, since we need to discard all traces before we can return **None**. We run them in order of the decreasing probability instead, as this will allow us to more quickly correct course in the rare case in which we have misclassified a passing trace as being in S_d: after running a few more copies of the trace, instead of discarding it, we would observe its empirical probability increasing, and we reclassify it into S_c on the next step.

If there is space left in the schedule after scheduling S_c, S_d and S_f as described, we add additional runs of the traces in S in a round–robin fashion, that is, copies beyond the expected number of executions required to 'prove' or 'disprove' each T_j, but without running any T_j more than nr times.

We show the execution of our heuristic on our same example from the previous two techniques:

$$[(0,0),(0,0),(0,0)] \xrightarrow[5,5,5]{} [(4,1),(2,3),(5,0)]$$
$$[(4,1),(2,3),(5,0)] \xrightarrow[2,0,13]{} [(6,1),(2,3),(18,0)]$$

3.3.3 Solving Trace Selection as an MDP

We can formulate the problem of trace selection as a Markov Decision Process (MDP), which allows us to solve for the optimal strategy, for given values of the parameters n, m, nr, st.

A Markov Decision Process is a tuple $(\mathbb{S}, \mathbb{A}, P, \gamma, R)$, where: $\mathbb{S}$ is a set of states, $\mathbb{A}$ is a set of actions and $P : \mathbb{S} \times \mathbb{A} \to \{Pr[\mathbb{S}]\}$ is a map from every state and action pair to a probability distribution over possible state transitions. $R : \mathbb{S} \to \mathbb{R}$ is the reward function, which associates a value with reaching each state. Finally, $\gamma \in [0, 1]$ is called the discount factor.

To execute an MDP, we start from some initial state $\sigma_0 \in \mathbb{S}$, then choose an action $a_0 \in \mathbb{A}$. The MDP then transitions to a state σ_1 chosen at random over the probability distribution $P(\sigma_0, a_0)$. The process is then repeated, selecting a new a_i for each σ_i and choosing σ_{i+1} from the distribution $P(\sigma_i, a_i)$. The value of the execution of the MDP is then $\sum_i \gamma^i R(\sigma_i)$, which is the reward of each state visited, multiplied by γ^i. The discount factor γ is used to decrease the reward of reaching "good" states, in proportion to how late in the execution these states are reached.

Given a policy $\pi : \mathbb{S} \to \mathbb{A}$, which is simply a mapping from states to actions, we calculate the value of the policy as the expected value $V^\pi(\sigma_0) = E[\sum_i \gamma^i R(\sigma_i)]$ where σ_0 is the initial state, and for every $i \geq 0$, $a_i = \pi[\sigma_i]$ and σ_{i+1} is chosen from $P(\sigma_i, a_i)$. Solving an MDP is equivalent to

finding a policy π that maximizes $V^{\pi}(\sigma_0)$.

We encode trace selection as an MDP as follows:

- $\mathbb{S} = \{[(s_0, f_0), ..., (s_{n-1}, f_{n-1})]\}$ is the set of possible combinations of observed values of (s_j, f_j) for each $T_j \in S$. The initial state is $\sigma_0 = [(0,0), ..., (0,0)]$
- $\mathbb{A} = \{[v_0, ..., v_{n-1}] \mid \sum_{j=0}^{n-1} v_j \leq m\}$ is the set of possible schedules *sch*.
- $P(\sigma_i, a)[\sigma_j]$ where $\sigma_i = [(s_{i0}, f_{i0}), ...]$, $a = [v_0, ...]$ and $\sigma_j = [(s_{j0}, f_{j0}), ...]$ with $\forall k.\, (s_{jk} + f_{jk}) - (s_{ik} + f_{ik}) = v_k$ is:
$$\prod_{k=0}^{n-1} \sum_{p} \binom{v_k}{s_\delta} p^{s_\delta} (1-p)^{(v_k - s_\delta)} Pr[P_{T_k} = p]$$
with $s_\delta = s_{jk} - s_{ik}$. $P(\sigma_i, a)[\sigma_j]$ is 0 if $\exists k.\, (s_{jk} + f_{jk}) - (s_{ik} + f_{ik}) \neq v_k$.
- $R(\sigma')$ is -1 for every state σ', and $\gamma = 1$

We make the reward negative and the discount factor 1, since we wish to find only a policy that minimizes the number of reached states, which is equivalent to minimizing the number of steps in `get_passing()`. Any state containing (s_j, f_j) with $s_j \geq st$ for any j, as well as any state where $f_j > nr - st$ for every j, is a terminal state of the MDP: once such a state is reached, the execution of the MDP ends.

Note too that the precise definition of $P(\sigma_i, a)[\sigma_j]$ requires knowledge of $Pr[P_{T_k} = p]$ for each subtrace T_k. But we have no way of precisely knowing this distribution. In practice, if we have a prior (discrete) probability distribution $Pr[P_{T'} = p]$ over the set of all possible subtraces, we can approximate:

$$Pr[P_{T_k} = p] = \sum_{p} \left(\binom{s_k + f_k}{s_k} p^{s_k} (1-p)^{f_k} \right) Pr[P_{T'} = p]$$

In Section 3.4, we approximate the prior by running `ND3MIN()` with naive round–robin trace selection on a few (app, trace, activity) tuples. We use that experimentally discovered prior to approximate $P(\sigma_i, a)[\sigma_j]$.

There are methods for solving an MDP in the general case, but they require iterating multiple times over the set $\mathbb{S}$ and calculating the expected value $V^{\pi}(\sigma')$ of each state based on the value of other states, until a fix-point is reached. In our formulation, the number of states, enumerated naively, is:

$$|\mathbb{S}| = \left(\sum_{k=0}^{k \leq nr} (k+1) \right)^n = \left(\frac{(nr+1)(nr+2)}{2} \right)^n$$

since we can construct $k+1$ pairs (s_j, f_j) with $s_j + f_j = k$. For $n = 5$ ($nr = 20$), this gives us over 6×10^{11} states.

We can significantly optimize our solution for this particular MDP in two ways, however: by getting rid of the fix-point iteration requirement (changing it to a single pass over $\mathbb{S}$), and by reducing the size of $\mathbb{S}$ itself. As we will see in a moment, this single pass still produces the optimal solution, since there is an indexing of the states in $\mathbb{S}$ such that the value of each state depends only on the values of states with a higher index, and our single pass traverses the states in decreasing order of their indices.

First, we observe that in our MDP we can never visit the same state twice. In fact, our MDP is a DAG, where each state must transition to one in which the sum of the elements of the tuples (s_j, f_j) has a higher value. This type of MDP is called a finite horizon MDP [60].

We now restrict all candidate policies π to include only schedules for which $\sum_{j=0}^{n-1} v_j = m$ exactly, except in the case in which doing so would violate the constraint $\forall j.\, s_j + f_j \leq nr$. In the latter case, every π always chooses to schedule as many runs of each T_j as needed to reach nr. We note that this last action must always lead to a final state.

Every state $\sigma' = [(s_0, f_0), ..., (s_{n-1}, f_{n-1})]$ reachable from the initial state σ following any π with the above restrictions, is either a final state or is such that $\sum_{j=0}^{n-1}(s_j + f_j) \mod m = 0$. We then define I, the state's iteration, such that $I[\sigma'] = \left(\sum_{j=0}^{n-1}(s_j + f_j)\right)/m$ for every non-final state. $I[\sigma']$ is always an integer. We assign all final states to iteration $I[\sigma'] = \lceil \frac{nr \times n}{m} \rceil$. We note that for every $\sigma_i \in \mathbb{S}$, π as above, and $\sigma_j \in P(\sigma_i, \pi[\sigma_i])$ we have $I[\sigma_j] > I[\sigma_i]$.

Given the partition of $\mathbb{S}$ into subsets $\mathbb{S}_i = \{\sigma \in \mathbb{S} \mid I[\sigma] = i\}$ induced by I, we can solve the MDP by visiting each $\mathbb{S}_i$ once in reverse order of i and computing the optimal $\pi[\sigma']$ and $V^\pi(\sigma')$ for each $\sigma' \in \mathbb{S}_i$. We do not need to iterate to reach a fix-point, since

$$V^\pi(\sigma') = R(\sigma') + max_{a \in \mathbb{A}} \sum_{\sigma''} \left(P(\sigma', \pi[\sigma'])[\sigma'']\right) V^\pi(\sigma'')$$

depends only on the values $V^\pi(\sigma'')$ of states reachable from σ' by actions in π. Since all such states satisfy $I[\sigma''] > I[\sigma']$, their value has already been calculated when visiting a previous $\mathbb{S}_{j>i}$. The time to solve the MDP by this method is $O\left(|\mathbb{A}|\,|\mathbb{S}|\right)$. Note that within a single $\mathbb{S}_i$, the problem of computing $V^\pi(\sigma')$ for the states in $\mathbb{S}_i$ is highly parallelizable.

We can further reduce $|\mathbb{S}|$ by merging states which are isomorphic in our model. First, we can coalesce all final states into two: a success state σ_s whenever $s_j \geq st$ for any j, and a failure state σ_f when $f_j > nr - st$ for every j. Furthermore, if state σ' has $f_j > nr - st$ for any j, this is equivalent (for the purposes of `get_passing()`) to the same state after replacing (s_j, f_j) with (X, Y) with $Y > nr - st$. We pick a single state in this equivalence set as representative, taking care to choose one which preserves the invariant $\sum_{j=0}^{n-1} s_j + f_j \mod m = 0$ and falls into the latest possible iteration. We can also reorder the tuples (s_j, f_j) in every state, since the order of the subtraces in S does not affect the result of `get_passing()`, and the generated policy π will be identical, up to a reordering of $a' = \pi[\sigma']$. After applying all optimizations, for $n = 5$, $m = 15$, $nr = 20$ and $st = 18$,

our algorithm must examine $|\mathbb{S}'| = 729,709$ states in order to compute the optimal strategy π.

After pre-computing π for a particular set of parameters n, m, nr, st, we can use it to generate a schedule *sch* at each step of the trace selection process. Section 3.4 compares the number of steps required by `get_passing()` when using this strategy versus our heuristic from Section 3.3.2.

3.4 Results

This section presents our empirical evaluation and results. Section 3.4.1 describes our experimental setup and presents the data discussed in the rest of the section. Section 3.4.2 explores the size of the minimized subtraces and the number of calls to `get_passing()` performed by the algorithm. Section 3.4.3 contrasts the performance of the different trace selection methods of Section 3.3. Finally, Section 3.4.4 explores the prevalence of application non-determinism in our dataset.

Our tests were performed in an Amazon EC2 cloud environment, in which we ran $m = 15$ Android emulator instances, each on its own virtual machine, as our test oracles. As mentioned before, the rest of the parameters used for `ND3MIN()` are $n = 5$, $nr = 20$ and $st = 18$.

3.4.1 Datasets

We evaluate our GUI event trace minimization approach on two different datasets: one composed of 7 applications from the Google Play store selected among the top 20 most popular apps in their category across different app categories (**gplay**), and another of 4 redistributable open-source applications from the F-droid open-source application repository (**fdroid**). For each application, we generated 10 random Monkey traces, of 500 events each, restricting ourselves to tap events exclusively. Although our approach can potentially be used to minimize traces with any type of events, reaching activities requiring events other than taps, such as structured keyboard input or complex touch gestures, often produces an unfeasibly large original trace when performing random testing alone.

We ran these traces on the corresponding application and recorded the activities reached during their execution. For the Google Play apps, we arbitrarily selected an average of 4 such activities per app, with the minimum being 3 activities. For the F-Droid dataset, we selected an average of 5.25 activities per app, with a minimum of 3. For each app A, and activity a, we took a trace T in our 10 generated traces which reached a with $P_T \geq 0.9$ (observed over 20 trace replays). In the cases where many traces reached a with equal probability, we picked one at random.

We first ran `ND3MIN()` on nine activities from two apps (`com.eat24.app` and `com.duolingo`) of the **gplay** set, using the naive (round–robin) trace selection strategy of Section 3.3.1. This produced calls to our test oracle O with 402 distinct subtraces in total. We used the oracle responses obtained from this preliminary experiment to generate a probability prior $Pr[P_{T'} = p]$ given an unknown T' and a probability p. We restricted ourselves to only two apps and nine activities for generating this

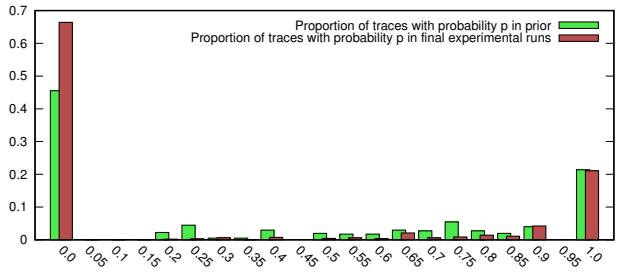


Figure 3.5: Prior probability distribution of subtraces.

prior for two reasons. First, minimizing traces under the naive strategy is a time consuming process, compared with either the heuristic or MDP methods. Second, we want to show that the prior computed for a few apps generalizes over unrelated Android applications, meaning that computing this prior is a one time cost, leading to a solution for the MDP model that can be applied without changes to minimizing traces for unknown apps.

Figure 3.5 shows this prior probability distribution. As observed in Section 3.3.3, we can use this prior to approximate the transition probabilities used in the MDP for computing the optimal policy for trace selection. To check the quality of this prior, we also plot the probability distribution as estimated by examining all queries to the test oracle performed by the rest of the experiments in this section (from a total of 3490 subtraces). This distribution shows more traces as always failing ($P_{T'} = 0$) and fewer traces as having low but non-zero probability. Otherwise, the distribution looks very similar to our prior, increasing our confidence in using said prior as our estimate of $Pr[P_{T'} = p]$.

For each dataset (**gplay** and **fdroid**) we then ran `ND3MIN()` on each tuple (A, T, a) in the set, under two different configurations for trace selection (see Section 3.3):

- One using the heuristic in Section 3.3.2 exclusively for every invocation of method `get_passing()`.
- One using the pre-computed optimal policy (Section 3.3.3) when `get_passing()` receives a set S with 2 to 5 subtraces (the values of n for which we are able to compute the optimal policy in reasonable time[2]). In this configuration, `get_passing()` defaults to using the same heuristic of the previous configuration, whenever `get_passing()` is passed $n \geq 6$ subtraces.

On average 66% of the steps executed in the MDP based trace selection case are steps in which `get_passing()` was invoked with fewer than 6 subtraces, and thus uses the optimal policy, based on solving the MDP. The remaining 34% fall back to using the same heuristic of Section 3.3.2.

Table 3.1 lists the apps and activities that constitute both of our datasets, and assigns a reference key to each app-activity pair. We use those keys to refer to the corresponding trace minimization experiments through the rest of this chapter.

[2]Solving the MDP using 4 2-vCPU EC2 VMs takes 8 min for $n = 3$, 43 min for $n = 4$ and 83 hours for $n = 5$.

Table 3.1: Dataset Key: App-activity pairs

Google Play Apps		
Key	Application	Activity
1-1	com.eat24.app	SplashActivity
1-2		HomeActivity
1-3		LoginActivity
1-4		CreateAccountActivity
2-1	com.duolingo	LoginActivity
2-2		HomeActivity
2-3		WelcomeFlowActivity
2-4		SkillActivity
2-5		LessonActivity
2-6		FacebookActivity
3-1	com.etsy.android	HomescreenTabsActivity
3-2		CoreActivity
3-3		DetailedImageActivity
4-1	com.ted.android	SplashScreenActivity
4-2		MainActivity
4-3		TalkDetailActivity
4-4		VideoActivity
4-5		BucketListInfoActivity
5-1	com.zhiliaoapp.musically	SignInActivity
5-2		OAuthActivity
5-3		TermOfUsActivity
6-1	com.pandora.android	SignUpActivity
6-2		SignInActivity
6-3		ForgotPasswordActivity
7-1	com.google.android	LicenseActivity
7-2	.apps.photos	LicenseMenuActivity
7-3		SettingsActivity
7-4		PhotosAboutSettingsActivity

F-Droid Apps		
Key	Application	Activity
8-1	com.evancharlton.mileage	Mileage
8-2		VehicleStatisticsActivity
8-3		TotalCostChart
8-4		FillupInfoActivity
8-5		FillupActivity
8-6		FillupListActivity
8-7		MinimumDistanceChart
8-8		AverageFuelEconomyChart
9-1	de.delusions.measure	MeasureTabs
9-2		MeasureActivity
9-3		BmiTableActivity
9-4		BmiCalcActivity
10-1	org.liberty.android	AnyMemo
10-2	.fantastischmemo	OptionScreen
10-3		AlgorithmCustomizationScreen
10-4		StudyActivity
10-5		CardEditor
10-6		SpreadsheetListScreen
11-1	org.totschnig.myexpenses	CalculatorInput
11-2		MyExpenses
11-3		ExpenseEdit

Table 3.2: Results for the Google Play apps

	heuristic				opt:mdp			
Key	events	steps	time	check	events	steps	time	check
1-1	0	4	4:34:01	20 (0.99/1.0)	0	4	4:17:47	20 (0.99/1.0)
1-2	0	4	3:45:04	20 (0.99/1.0)	0	4	5:48:18	20 (0.99/1.0)
1-3	3	18	13:27:12	20 (0.99/0.94)	3	17	12:22:16	20 (0.99/0.94)
1-4	5	51	35:17:52	19 (0.94/0.91)	5	45	32:33:16	19 (0.94/0.91)
2-1	0	4	3:34:23	20 (0.99/1.0)	0	4	4:19:35	20 (0.99/1.0)
2-2	2	21	11:11:34	20 (0.99/0.96)	2	15	12:51:19	20 (0.99/0.96)
2-3	2	18	12:06:37	20 (0.99/0.96)	2	16	13:11:39	20 (0.99/0.96)
2-4	19	73	40:46:06	18 (0.69/0.72)	16	62	51:01:40	12 (0.00/0.76)
2-5	25	87	50:16:46	11 (0.00/0.65)	23	110	87:56:06	11 (0.00/0.67)
2-6	2	64	52:14:13	20 (0.99/0.96)	7	85	48:02:17	20 (0.99/0.88)
3-1	0	4	3:18:15	20 (0.99/1.0)	0	4	3:27:18	20 (0.99/1.0)
3-2	3	28	21:27:39	20 (0.99/0.95)	3	71	43:27:56	20 (0.99/0.95)
3-3	8	50	38:55:17	15 (0.11/0.87)	7	35	26:31:25	7 (0.00/0.88)
4-1	0	4	5:27:51	20 (0.99/1.0)	0	4	4:11:50	20 (0.99/1.0)
4-2	1	13	8:28:01	20 (0.99/0.98)	1	10	7:04:49	20 (0.99/0.98)
4-3	7	31	17:45:40	19 (0.94/0.88)	7	24	15:59:37	20 (0.99/0.88)
4-4	15	57	39:02:54	19 (0.94/0.77)	11	32	22:27:27	18 (0.69/0.82)
4-5	5	13	13:45:54	20 (0.99/0.91)	5	11	10:26:14	19 (0.94/0.91)
5-1	2	16	13:52:34	20 (0.99/0.96)	2	16	15:04:38	20 (0.99/0.96)
5-2	1	13	14:32:24	20 (0.99/0.98)	1	14	13:55:28	20 (0.99/0.98)
5-3	1	12	12:53:04	20 (0.99/0.98)	1	13	12:44:12	20 (0.99/0.98)
6-1	1	18	21:12:14	20 (0.99/0.98)	1	14	16:38:10	20 (0.99/0.98)
6-2	1	13	18:32:25	20 (0.99/0.98)	1	14	15:48:11	20 (0.99/0.98)
6-3	8	72	61:41:23	18 (0.69/0.87)	3	49	47:00:46	17 (0.43/0.94)
7-1	6	46	51:34:31	20 (0.99/0.90)	5	47	40:44:01	18 (0.69/0.91)
7-2	5	50	51:35:34	19 (0.94/0.91)	5	46	39:18:00	18 (0.69/0.91)
7-3	3	27	34:18:45	20 (0.99/0.94)	2	36	31:02:26	20 (0.99/0.96)
7-4	3	34	32:21:50	20 (0.99/0.94)	4	34	28:26:12	20 (0.99/0.93)
Average	4.57	30.18	24:34:17		4.18	29.86	23:48:40	
Median	2.5	19.5	18:09:03		2.5	16.5	15:53:54	

Table 3.3: Results for the F-Droid apps

	heuristic				opt:mdp			
Key	events	steps	time	check	events	steps	time	check
8-1	0	4	3:09:23	20 (0.99/1.0)	0	4	3:06:46	20 (0.99/1.0)
8-2	2	17	8:35:44	20 (0.99/0.96)	2	15	8:48:25	20 (0.99/0.96)
8-3	3	25	12:46:05	20 (0.99/0.95)	3	25	13:16:33	20 (0.99/0.95)
8-4	10	84	41:21:14	20 (0.99/0.84)	10	85	36:39:12	20 (0.99/0.84)
8-5	0	4	3:12:27	20 (0.99/1.0)	0	4	3:31:56	20 (0.99/1.0)
8-6	2	16	9:09:12	19 (0.94/0.96)	2	21	10:58:12	19 (0.94/0.96)
8-7	3	15	7:46:58	20 (0.99/0.95)	3	14	11:07:33	20 (0.99/0.95)
8-8	3	14	7:39:26	20 (0.99/0.95)	3	14	8:09:53	20 (0.99/0.95)
9-1	0	4	3:35:59	20 (0.99/1.0)	0	4	3:23:09	20 (0.99/1.0)
9-2	0	4	3:12:24	20 (0.99/1.0)	0	4	2:56:53	20 (0.99/1.0)
9-3	2	17	8:52:07	20 (0.99/0.96)	2	26	11:32:35	20 (0.99/0.96)
9-4	4	48	22:47:29	20 (0.99/0.93)	3	27	13:01:08	20 (0.99/0.93)
10-1	0	4	3:20:17	20 (0.99/1.0)	0	4	3:12:32	20 (0.99/1.0)
10-2	3	16	8:18:16	20 (0.99/0.95)	3	14	8:06:41	20 (0.99/0.95)
10-3	5	61	36:57:50	20 (0.99/0.91)	5	51	30:44:08	20 (0.99/0.91)
10-4	2	17	10:29:14	20 (0.99/0.96)	2	17	9:40:59	20 (0.99/0.96)
10-5	6	49	24:30:42	18 (0.69/0.90)	6	35	16:52:17	20 (0.99/0.90)
10-6	3	29	19:02:10	20 (0.99/0.95)	3	40	24:33:01	20 (0.99/0.95)
11-1	0	4	3:24:06	20 (0.99/1.0)	0	4	3:19:23	20 (0.99/1.0)
11-2	3	52	26:12:57	20 (0.99/0.95)	3	43	21:40:33	20 (0.99/0.95)
11-3	13	70	32:22:06	16 (0.23/0.80)	11	66	36:05:57	17 (0.43/0.82)
Average	3.05	26.38	14:07:55		2.9	24.61	13:22:16	
Median	3	17	8:52:07		3	17	10:58:12	

For the applications in the Google Play dataset, Table 3.2 shows the size of the minimal subtrace obtained by our minimization algorithm for each target activity, together with the number of steps and wall-clock time that our algorithm took to extract it in each configuration. Table 3.3 shows the analogous information for the F-Droid open-source apps. The performance of the naive method on the 2 apps of the **gplay** set, used to compute the $P_{T'}$ priors, is listed in Table 3.4.

3.4.2 Size of Minimized Traces, Steps and Check

The column labeled `events` in our tables specifies the number of events in the minimized trace T_{min} as generated by our algorithm. Recall that our input traces are 500 events long in each experiment. The average length of the minimized traces is 4.57 for the **gplay** dataset using the heuristic trace selection, 4.18 using the MDP-based version. For the **fdroid** dataset, these numbers are 3.05 and 2.9, respectively. The fact that only a few events are needed in each case to reach the desired activity shows the value of minimization. Our minimized traces are smaller than the original Monkey traces by an average factor of roughly 100x.

Perhaps another take away from these numbers is that running many rounds of random monkey testing for a few events, or even systematically testing many small sequences sampled from a large set of events, might prove at least as valuable for uncovering app behavior as a few much longer monkey runs. However, note that there is a fixed cost in reseting the application or the emulator, which quickly becomes the dominating factor when running small event traces. Additionally, true

Table 3.4: Performance for naive trace selection

	naive			
Key	# events	# steps	exec. time	check
1-1	0	8	7:00:20	20 (0.99/1.0)
1-2	0	8	6:41:37	20 (0.99/1.0)
1-3	3	50	37:22:56	20 (0.99/0.95)
1-4	4	66	48:01:59	17 (0.43/0.93)
2-1	0	8	6:53:19	20 (0.99/1.0)
2-2	2	26	14:07:33	20 (0.99/0.96)
2-3	2	27	15:47:21	18 (0.69/0.96)
2-4	15	86	62:55:47	15 (0.11/0.77)
2-5	30	80	62:41:16	14 (0.04/0.59)
Average	6.2	39.9	29:03:34	
Median	2	27	15:47:21	

exhaustive testing remains infeasible. Assuming 798 distinct events to chose from (a number based on the automated testing system we shall discuss in Chapter 4), testing all 4 event sequences means starting the app $798^4 = 405 \times 10^9$ times.

The column labeled `steps` counts the number of times the method `get_passing()` generated a schedule and called our 15 test oracles in parallel. Equivalently, `steps` is the maximum number of sequential calls to each of the test oracles required during our minimization algorithm. To make sure the trace is suitable for minimization, our implementation first runs the original trace T 20 times, and aborts running if the oracle doesn't accept T in at least 15 of those calls. We include the two steps required for this check in our count.

The column labeled `check` contains a triplet of the form c (p_1/p_2). After our minimization algorithm has finished, we run the resulting trace 20 additional times, and record as c the number of times it succeeds. We use this number to calculate $p_1 = Pr[P_{T_{min}} \geq 0.85]$ and p_2, the probability that, if $P_{T_{min}} \geq 0.85$, then T_{min} is approximately 1-minimal, as defined by Lemma 1. For p_1, we use a formula analogous to that of Section 3.2, but taking into account the exact number of successful oracle queries:

$$p_1 = \frac{\sum_{p \geq 0.85} \binom{nr}{c} \cdot p^c (1-p)^{nr-c} Pr[P_{T'} = p]}{\sum_p \binom{nr}{c} \cdot p^c (1-p)^{nr-c} Pr[P_{T'} = p]}$$

For the probability prior required for these calculations, we use the same prior from Figure 3.5 used by the MDP trace selection method. All probabilities in the tables are truncated, not rounded, as we wish to obtain a lower bound. As observed in Section 3.2, since the error of `passes()` accumulates through multiple trace reductions in our algorithm, our final $P_{T_{min}}$ can fall below p_b, so it is not always true that $c \geq 18$. The vast majority of our minimized traces fulfill $P_{T_{min}} \geq 0.9$, and all but one succeed over 50% of the time.

Note that for the same (app, trace, activity) tuple, our algorithm sometimes produces minimized traces of different sizes when using different trace selection strategies. This can happen for one

of two reasons. First, different trace selection strategies cause delta debugging to pick different subtraces during recursive invocations of the `MinR()` method, which can guide the algorithm towards discovering different 1-minimal solutions. A 1-minimal solution does not imply the returned trace is of minimum length among all possible successful subtraces, and an input trace can contain multiple distinct 1-minimal subtraces that reach the desired activity and have different lengths. Because of the probabilistic nature of our algorithm, it is also possible that the trace returned by `ND3MIN()` is not truly 1-minimal, especially if it contains a subtrace which reaches the activity with a probability very close to 0.9, which our algorithm might have trouble classifying either way.

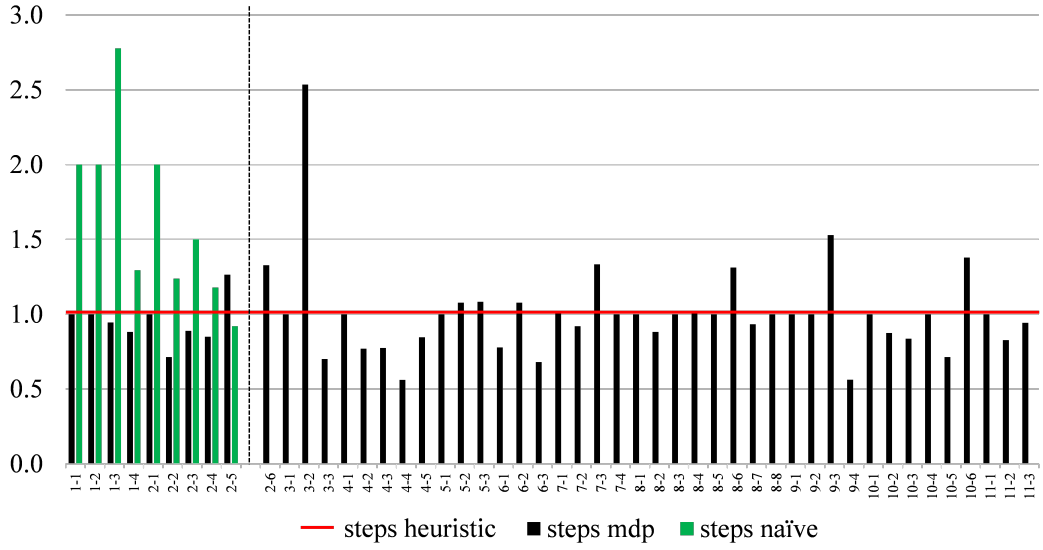
We can observe this situation when contrasting the minimized traces discovered by the naive and heuristic trace selection methods for `CreateAccountActivity` in `com.eat24.app` (experiment 1-4). The heuristic method produces a 5 event minimized trace that passes 19 out of 20 oracle calls in its final check, while the naive method returns a 4 event subtrace, which is actually a subtrace of the one returned by the heuristic case, but which only passes 17/20 checks. We re-ran both traces 300 times, which suggests the underlaying probability of the 5 GUI event trace is ≈ 0.89 and that of the 4 GUI event trace is ≈ 0.84.

3.4.3 Performance Comparison

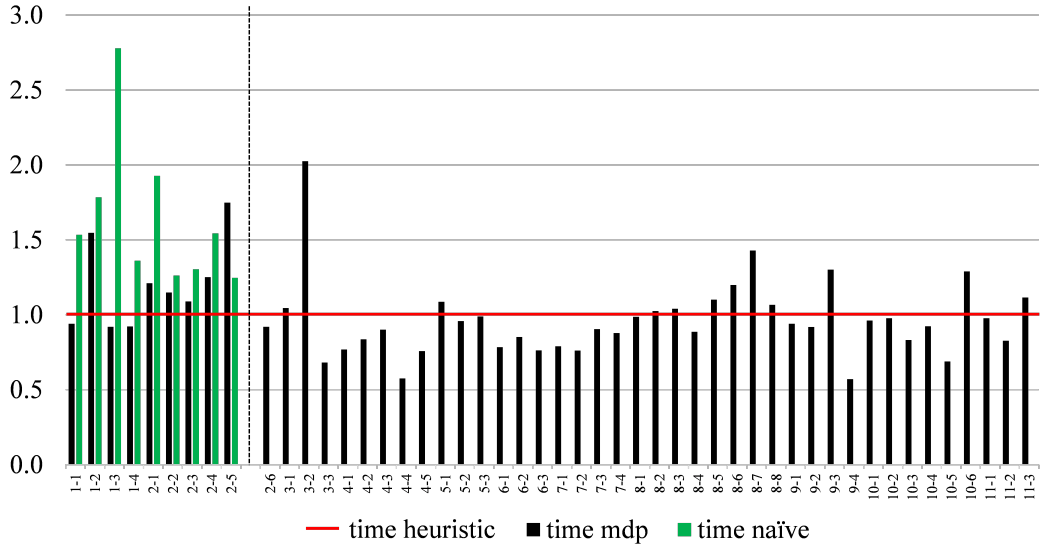
Figure 3.6 plots the number of steps and wall-clock time for each experiment, comparing the naive, heuristic and MDP-based trace selection methods. We normalize in each experiment to the value obtained in the heuristic case, since we only have the performance of the naive method for the limited set of (app, trace, activity) tuples in Table 3.4. Thus the red line at 1 represents the performance of our heuristic, and the bars represent the performance of the naive and MDP methods relative to that of the heuristic. The dashed vertical line separates the experiments for which we have data on the naive method from those for which we don't.

We note that the configuration using the MDP policies doesn't always outperform our heuristic. This is not unreasonable, since: a) the MDP method only guarantees to minimize the number of steps in `get_passing()`, but it might pick different subtraces than other methods, thus failing to minimize the number of steps over the whole algorithm, b) even within a call to `get_passing()` we are approximating the prior of $P_{T'}$ based on previous experiments, which could lead to non-optimal results if the distribution of underlying trace probabilities is very different in the particular app under test, versus the set used to compute the prior, and c) since the oracle is non-deterministic, the number of steps our algorithm must perform, even when using the same trace selection method, varies across runs. Our results do indicate, however, that using the heuristic method for trace selection is a reasonable option, which performs similarly to solving the MDP formulation and avoids the expensive pre-computation for every value of n.

Two outliers in our plots bear explaining. In experiment 2-5, both our heuristic and the MDP based method seem to under-perform the naive method on the number of steps. However, this is also



(a) Normalized proportion of mdp and naive steps versus heuristic



(b) Normalized times of mdp and naive versus heuristic

Figure 3.6: Trace selection performance comparison

a case in which the naive method produces a larger trace (30 events) than either the heuristic (25 events) or the MDP based method (23 events), so this outcome can be explained as the result of the naive method having stopped the minimization process earlier than the other two. In experiment 3-2 the MDP based method performs much worse than our heuristic while producing a trace of identical size. In this case, the heuristic found a trace consisting of 3 consecutive events of the original trace, while the MDP based method found a trace of non-consecutive events. The structure of delta debugging is such that it generally makes faster progress towards a contiguous subsequence than towards a non-contiguous one.

The average running time of our minimization algorithm using the heuristic approach is 24:34 hours for the activities in the **gplay** set (median: 18:09 hours) and 14:08 hours for the **fdroid** set (median: 8:52 hours). Using the MDP based method, we have an average of 23:49h and median of 15:54h for the **gplay** set, and an average of 13:22h and median of 10:58h for **fdroid** apps. Thus, our approach fits comfortably in the time frame of software processes that can be run on a daily (i.e., overnight) or weekly basis.

We tested the sequence of time measurements for all apps (**gplay** and **fdroid**) under the Wilcoxon Signed-Rank Test [110] and found a mean rank difference between the heuristic and MDP based methods with $p \approx 0.08$, which is not quite enough to be considered statistically significant. We do not have enough samples under the naive method to compare it against the other two under a similar test, but it can be seen from Figure 3.6 that this method often significantly underperforms compared to the other two.

3.4.4 Effects of Application Non-determinism

One final question regarding the trace minimization problem is on whether or not handling application non-determinism is truly a significant problem. As we discussed in Section 3.1, some Android applications present non-deterministic behavior under the same sequence of GUI events, motivating the need for running each GUI event trace multiple times during minimization and estimating trace probabilities. However, if this is a problem that occurs only rarely, it might be that the techniques presented in this paper are not often required. We argue that application non-determinism is in reality a common problem, as can already be somewhat discerned from the fact that the check columns of tables 3.2 and 3.3 often show traces as succeeding in reaching the target activity less than 20 times in 20 runs.

To explore the impact of application non-determinism for trace minimization, we took the output minimized traces of our MDP condition for the **gplay** dataset, and attempted to further minimize them by using traditional non-probabilistic delta-debugging (i.e. by following the algorithm in [119] or, equivalently, by running `ND3MIN()` with $nr = st = 1$). In 8 out of 28 cases (29%), this produced a further reduced trace. We then ran each of these resulting traces an additional 20 times. Table

Table 3.5: Effect of application non-determinism in our dataset

Application	Key	Activity	non-deterministic mdp		+ deterministic DD	
			# events	check	# events	check
com.eat24.app	1-3	LoginActivity	3	20/20	2	6/20
	1-4	CreateAccountActivity	5	19/20	3	15/20
com.duolingo	2-4	SkillActivity	16	12/20	11	4/20
	2-5	LessonActivity	23	11/20	15	1/20
com.etsy.android	3-2	CoreActivity	3	20/20	2	12/20
com.ted.android	4-4	VideoActivity	11	18/20	10	1/20
com.google.android	7-2	LicenseMenuActivity	5	18/20	4	18/20
.apps.photos	7-4	PhotosAboutSettingsActivity	4	20/20	3	18/20
Average			8.75	17.25/20	6.25	9.38/20

3.5 contrasts the size and reliability of the traces minimized under the original MDP based non-determinism aware condition, with that of the result of further applying traditional delta debugging to these traces. As we can see, these resulting traces succeed in reaching the target activity much less frequently than the originally minimized traces. Thus, it is clear that for traces that succeed non-deterministically, it is important to take into account their corresponding success probabilities during minimization. This likely becomes more significant the more steps delta debugging takes, as we can see by looking at the cases of the table above in which the minimal trace obtained by the MDP strategy is larger than 5 events.

Chapter 4

Automated GUI Exploration based on Image Differences

In the previous chapter, we examined the problem of minimizing GUI event traces generated by an automated testing tool such as Android's Monkey. We noted that fully random app exploration, such as that performed by Monkey, produces large sequences of events, most of which do not play a part in triggering any interesting behavior (e.g. taps on inactive portions of the screen). To facilitate trace minimization, and in general to improve the performance of automated GUI testing, a smarter monkey – one which learns which input events to produce and which to avoid based on the current application state – is desirable.

There has been significant work in developing these more intelligent automated testing agents. These tools leverage approaches such as better detection of relevant system events [74], model-based testing [2, 3, 91, 115, 21, 11], concolic testing [6], evolutionary algorithms [75] and static analysis with application rewriting [15]. However, a recent survey paper by Choudhary et al. [22], which compares many of the tools above, seems to suggest that the standard Android Monkey remains competitive with regards to code coverage in a limited time, for a number of real world apps. One reason for this discrepancy might simply be the difference between the robustness expected from an industry standard tool and that of research prototypes. However, we believe that Monkey also gains two important advantages from its simplicity: the speed at which it can generate events, and its independence from the implementation details of the app under test. In this chapter, we introduce a technique for improving on pure random testing, while preserving this second advantage.

We implemented a different sort of intelligent monkey tool, which aims to be as insensitive to the app's implementation as possible. Unlike automated testing tools that rely on static or dynamic analysis of the application code, or on inspecting the running app using tools like the Android accessibility API, our approach interacts with the app under test exclusively by sending UI events

and taking screenshots of the current application state as would be visible to a human user. This approach is agnostic to any GUI toolkit, programing language or implementation technique used by the application. We should note that, for the purposes of our evaluation, we do instrument the apps to obtain method coverage information, which allows us to compare our approach against pure random testing. However, the tool itself requires no instrumentation to run on any existing app.

The tool generates taps on particular points of the screen, learning to avoid tapping on inactive areas. It remembers the appearance of a small screen patch around the tapped area and exploits regularities across space and time in the app's visual representation. Namely, it assumes that the same exact screen patch in the same location, at two different screen states or two different points in time, is likely to respond similarly to interaction. It also assumes that large portions of the screen that are visually identical will likely also share behavior. The tool determines whether the app reacts to a tap in a particular portion of the screen, the way a human user would, by observing whether or not the screen itself changes in response to such action. As we discuss later in this chapter, this approach has some advantages over looking at presumably more direct metrics of application activity, such as code execution. We focus exclusively on taps in our prototype, but, as with trace minimization, the approach can be extended to other GUI events.

4.1 Interaction Model

Figure 4.1 shows, at a high level, the way our tool interacts with the application under test, and the internal state it keeps. To the left, we show a particular screenshot of the app under test (this particular example uses a screen from `com.duolingo.app`). To the right, we show a representation of the state of the agent's memory.

The screen of the app under test is divided into a grid of squares of 50×50 pixels, and the agent has $21 \times 38 = 798$ actions available to it, corresponding to taps in the center of each of the 798 squares[1]. A semi-transparent overlay shows how our agent regards the expected likelihood of a tap in each square resulting in a response from the application: squares overlaid in dark gray are expected to be inactive, whereas squares overlaid in bright green are known to be active. Some squares have colors in between those two extremes, representing actions our agent is less confident about. The portions of the screen without any overlaid color are areas our agent considers unexplored. The faint blue dot on top of the "START" button represents the center of the square in which the agent intends to tap next. We selected 50×50 pixels squares, since any smaller targets in a mobile GUI would represent too small a tap target for a user operating a touch screen.

For each square in the grid that divides the screen, the agent's memory stores a list of 50×50 pixels image patches corresponding to what screen state it has seen in that particular position in the grid, associated with a confidence value that such square is clickable. To avoid unbounded memory

[1] For an app running on an emulator with 1080x1920 resolution.

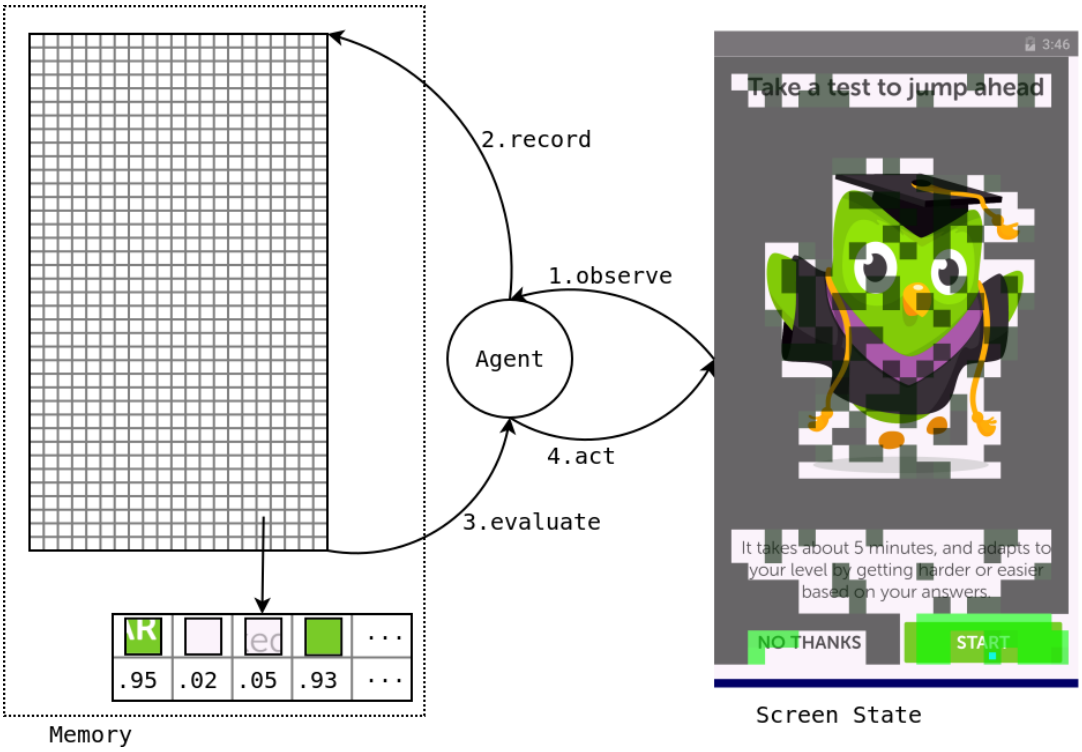


Figure 4.1: Overview of the tool's interaction model

usage, this list is limited to store no more than 200 versions of each square in the grid (corresponding to different screens within the app), evicting older squares if necessary. This proved to be enough in our experiments. Note that, although the diagram depicts actual image patches being stored in the memory, in reality we store only a hash of the corresponding image instead.

At the highest possible level, our agent operates by repeating a four step sequence: First, it observes the state of the screen and determines if the last action it took resulted in any response from the app. Second, it updates the expected response probability of the last square it tapped based on whether the app reacted to the action or not, recording that information into its internal memory. Third, it evaluates the available actions based on the state of the screen and its own memory, separating the possible squares into unexplored and explored squares, and assigning a probability of being clickable to those in the second group, in a manner similar to the screen state overlay in Figure 4.1. Finally, it chooses a particular square, and acts by tapping on the center of the square. After that, the agent goes back to the observation step.

4.1.1 Observe and record

At the beginning of every cycle, the agent checks if the screen state has changed in response to its previous action. If it has, it looks into its memory for the image patch corresponding to the square of the screen it tapped last. If no such image patch exists, it adds the patch to its memory with an initial probability of being clickable in the future, based on whether or not it was clickable this time: $p = 0.85$ if it was, $p = 0.15$ if it wasn't. If a record already exists in memory for this patch, then it is updated based on the following formula, where p' is the current probability of being clickable for that image patch in the same screen position, $c = 1$ if the square was clickable this time and $c = 0$ otherwise:

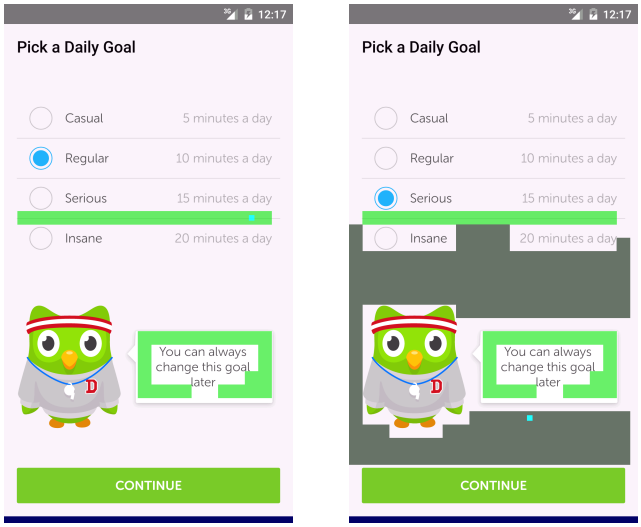
$$p = 0.5 * c + 0.5 * p'$$

Note that in these calculations, the image patches that get updated are always those corresponding to the screen state from before the corresponding action was performed, meaning those corresponding to the screen state in the previous iteration of the agent's four step cycle, rather than the current screen state.

A vital optimization to the scheme above is that we perform flood filling of adjacent and identical screen patches after an action. That is, whenever the agent taps into a square and receives a response (clickable or non-clickable), it updates its internal memory with that response not only for that particular square, but also for each of the four adjacent squares (left, right, above and below), as long as the image patch in those squares is identical to that of the tapped square. Then, it recursively performs this propagation on the neighbors of those four squares as well. Figure 4.2 shows the effect of flood filling after clicking on the inactive background of a particular app screen. Besides background detection, this optimization also exploits regularities in the appearance of large widgets (such as the edges of buttons) to increase the amount of screen state that the tool can consider explored with the same number of observe-update-evaluate-act cycles.

4.1.2 Evaluate and act

After updating its internal memory, the agent must decide on the next action to perform, namely, which of the 798 available squares of the screen to tap on. One question that naturally arises from this set up is when should the agent tap on an already explored square versus an unexplored one. Clearly, if our aim is to uncover novel behavior at the app level, then at least in the beginning we should always prefer new actions to those already tried, even those which we know get us a response from the app, since it is likely a response we have already seen. However, at the same time, tapping on known-active regions of the screen allows the agent to navigate around in the app, switching between different screens, which have different unexplored actions to be tried. To solve this problem, we actually allow the agent to automatically adjust its preference between the two



(a) Before **(b) After**

Figure 4.2: Flood-fill propagation of clickability info

depending on which approach is currently better at uncovering new behavior in the app at that particular point in time, based on new code coverage.

The agent keeps track of two values E_u and E_c, the time weighted sum of new coverage obtained by a tap on an unexplored or clickable square, respectively. Every time the agent performs an action that results in code coverage increasing (measured as method coverage in our experiments), it updates the corresponding value as follows, where cov_new is a value quantifying the coverage increase:

$$E_u = \text{cov\_new} + \gamma * E_u \text{ or}$$

$$E_c = \text{cov\_new} + \gamma * E_c$$

Here, γ is an adjustable discount parameter. In our experiments, $\gamma = 0.75$.

While $E_u \geq E_c$, the agent will elect to click on unexplored squares every time, as long as any unexplored area remains. However, when $E_u < E_c$, the agent will instead always click on known-clickable squares. Over time, if no new coverage is being found using either a "tap on unexplored" or "tap on known-clickable" strategy, then its corresponding weighted sum value will decrease relative to the other and the agent will switch its approach. Note that if the agent doesn't have access to code coverage information, we can use a different metric, such as the frequency with which tapping on an unexplored square results on finding a new clickable square, to decide when to switch between testing unexplored squares in the current screen, versus exercising known app behavior.

After the agent has decided between an unexplored or a known-clickable square, it must still

decide on the specific square to tap on within that category. We have no information to distinguish unexplored squares, so in that case we simply pick a square at random. When deciding to tap on a known-clickable square, however, we have different estimates on the probability of each such square actually resulting on a response from the app. We could sample squares at random, test them against a value generated uniformly at random between 0 and 1 and choose them as our next action if their estimated p is above said value. However, because of the way we update our clickability estimates, most squares, including those in the screen's background, will have some $p > 0$, and the standard situation is a screen with a few regions with values of p close to 1 and large swaths of screen real estate with values of $p \in [0.05, 0.15]$ which are usually fully unclickable. Because the low p regions are vast, most of the squares we test will end up coming from these regions. If we generate a new acceptance threshold for each square we examine and sample the squares at random, at the end of the process we are more likely to end up with one of the abundant low p squares rather than a square with a high probability of being clickable. Instead, we select an acceptance threshold $p_t \in [0, 1]$ (uniformly at random) first, then we chose at random between the set of squares with $p \geq p_t$.

Once a square to tap has been selected, the agent uses standard Android tools to send a tap event to the coordinates corresponding to the square, waits a brief interval for the screen to update, and goes back to the beginning of the cycle.

4.1.3 Putting it all together

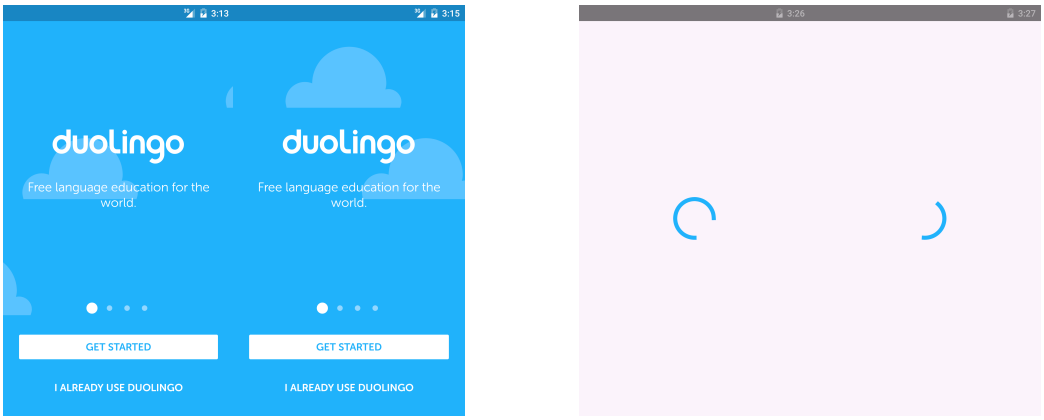
To recap, listing 4.3 shows the simplified pseudo-code for the agent's behavior.

After some initialization, the agent runs in a continuous loop. Line 9 takes a screenshot of the app under test, performing the observation step. The recording step (lines 12–23) covers checking if the screen changed and updating the internal state of the agent. Lines 13–18 update the p associated with the square tapped in the previous application screen, as well as those squares in its "flood fill neighborhood", as defined in Section 4.1.1. Note that here *last_action* represents the index associated with the tapped square, while *action* ranges over the indices of squares in the neighborhood, and memory acts as a two level map, indexed first by this action index, and then by a hash of the particular image patch whose value p it retains. Similarly, lines 19–23 look at whether the last action executed resulted in any new application coverage, and adjust the expected rewards E_u and E_c, depending on whether the last tap was in an unexplored or known clickable square.

The evaluation step is represented by lines 25–35. First, the agent decides whether to pick an unexplored or known clickable action, based on whether E_u or E_c is higher. If unexplored, it picks the specific square uniformly at random (lines 26–27). If known clickable, it first selects a random $p_t \in [0, 1]$ and then, if there are any available actions in its current view of the screen with $p \geq p_t$, it selects at random among those. If no such action exist, it selects a new, smaller, p_t and tries again (lines 28–35).

```
1  last_action ← None;
2  last_action_type ← None;
3  last_screenshot ← None;
4  E_u ← 0;
5  E_c ← 0;
6  memory ← {};
7  while true:
8      // 1. Observe;
9      screenshot ← take_screenshot();
10     // 2. Record;
11     if last_action:
12         v ← (screen_changed(screenshot) ? 1 : 0);
13         for (action, img_patch) ← flood_fill_neighborhood(last_screenshot,last_action):
14             l ← memory[action];
15             if hash(img_patch) ∈ l:
16                 l[hash(img_patch)] ← 0.5 × v + 0.5 × l[hash(img_patch)];
17             else:
18                 l[hash(img_patch)] ← 0.7 × v + 0.15;
19         cov_new = get_last_event_coverage();
20         if last_action_type = UNEXPLORED:
21             E_u = cov_new + γ ∗ E_u;
22         else:
23             E_c = cov_new + γ ∗ E_c;
24     // 3. Evaluate;
25     if E_u ≥ E_c:
26         action ← random({a ∈ memory | screenshot[a] ∉ memory[a]});
27         last_action_type ← UNEXPLORED;
28     else:
29         valid_acts = {};
30         p_t ← 1;
31         while empty(valid_acts):
32             p_t ← random({0, p_t});
33             valid_acts = {a ∈ memory | screenshot[a] ∈ memory[a] ∧ memory[a][hash(screenshot[a])]
                 ≥ p_t};
34         action ← random(valid_acts);
35         last_action_type ← CLICKABLE;
36     // 4. Act;
37     send_to_emulator(action);
38     last_action ← action;
39     last_screenshot ← screenshot;
```

Figure 4.3: Agent's operation



(a) Example A: Background animation **(b) Example B: Spinner**

Figure 4.4: Automatically changing application screen state

Finally, the agent acts by performing the tap action on the selected square in the emulator (line 37), and sets aside the current action and screenshot for the next iteration of its loop.

4.2 Detecting application activity

In the previous section, we mentioned that our agent detects whether its last action produced any effect on the app under test by observing whether the image on the screen changed after the action. The basic way we detect this is by comparing the screenshot taken after the previous action to that taken after the current action, by subtracting the pixel values of the first to those of the second. However, some care must be taken to make sure any difference is indeed due to the action just executed.

First of all, we must introduce a delay between actions, to allow the application to respond to the previous action. Otherwise, if we feed the application multiple taps in quick succession and observe a change in the screen, we might not be able to differentiate which action caused the change. In our prototype, we introduce a four second delay between actions, which we found necessary when running apps on top of the default Android emulator inside an Amazon EC2 VM. This means that our agent has a lower throughput of GUI events compared to a tool like Android Monkey, which can produce events at a significantly faster rate, often ignoring their effects on the app until a crash or a particular response is observed. However, note that the need to delay between actions is shared by any automated testing tool which must examine the state of the application after every action, and treat its internal operation in a purely blackbox manner. Relaxing this requirement via some form of lightweight and robust runtime inspection would be an useful refinement for this technique as a whole.

Second, there are a number of cases in which two screenshots of the same app, taken a few seconds apart, will be different, even if no user event has induced a changed in the application state. Apps include GUI widgets that change appearance constantly, without user intervention, such as animations, spinners, or even the system's clock. Figure 4.4 shows a few example screens before and after a tap in an inactive portion of the screen, where nonetheless the screen state has changed.

To compensate for this case, we actually take two screenshots between every two actions, one at the exact midpoint S_m and one right at the end of the interval, before the next action, S_e. We first look at the difference between S'_e, the end screenshot from the previous interval, right before the current action a, and S_m, to determine whether the screen changed in response to a, but subtract squares of the screen we know to be changing without GUI actions. We keep track of such automatically changing portions of the screen by looking at the differences between S_m and S_e, marking those 50×50 squares (as in the tap grid of the agent) as changing without GUI actions. Since the portions of the screen that are animated change over time, we decay this information slowly. We keep two maps of constantly changing screen squares, only writing newly found changing squares to the first, but acknowledging also those marked in the second. Every ten cycles of the agent, we discard the second map, promote the first to the position of the second and create a new empty map (meaning no square is considered automatically changing within it) for the first. Additionally, if the maps would ever result in more than 70% of the screen being ignored as constantly changing, we clear both immediately. This heuristic deals with the fact that splash screens and slow transitions between app screens sometimes cause the entire screen to be marked as changing.

We selected image difference after trying a number of metrics related to method activity on the app under test. One such alternative was to look at the number of methods, regardless of whether they had been covered before or not, which executed soon after the point in time at which the agent performed an action. Besides avoiding the requirement for application instrumentation, we chose image difference because we found it to be less noisy in practice than method-execution based metrics. We found that many real world apps perform significant computation in the background, unrelated to user-triggered GUI events. The noise from this sort of computation was variable enough to drown a signal derived from looking only at the number of app method invocations per second. Although we could have tried to discriminate between background methods and GUI-triggered methods by using an analogous technique to that which we use to detect autonomously changing screen state, we found image difference to be in general a less noisy alternative to begin with, compared to method execution, as a metric of the app's response to GUI actions.

4.3 Evaluation

In this section, we evaluate the performance of the agent described in this chapter. We compare it against a baseline version of the agent that produces taps uniformly at random in the same

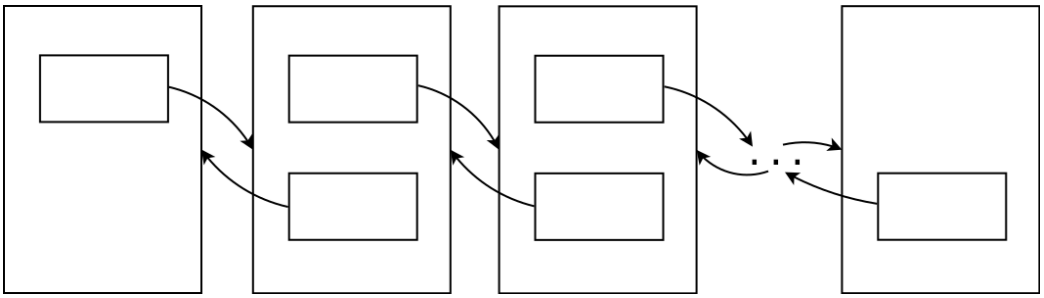


Figure 4.5: Simulated App 1: Back/Next Screens

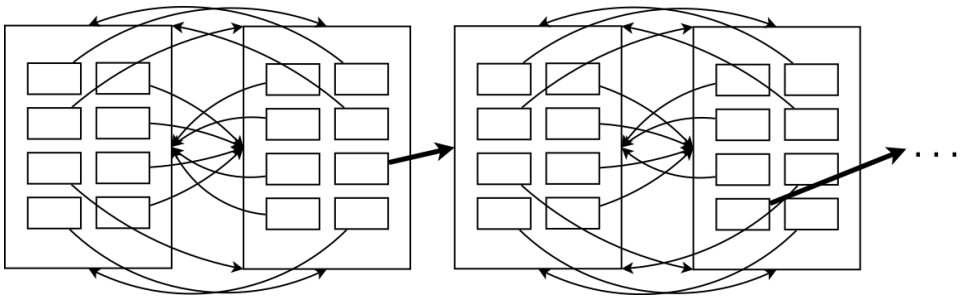


Figure 4.6: Simulated App 2: Tangled pairs

grid positions of the screen that the agent considers as valid actions. In Section 4.3.1 we present three separate simulated applications that reward the agent for reaching distinct screens within the simulated app, and compare both agents on the reward they are able to achieve over a certain number of actions. In Section 4.3.2 we compare both agents on a set of real world applications taken from the Google Play store. Our evaluation set-up instruments the apps under test using the *Ella* [5] tool, to produce continuous method coverage information. We compare the agents based on method coverage achieved over number of actions for these apps.

Finally, in Section 4.3.3, we discuss a few additional metrics related to our agent, such as the number of distinct image patches it remembers by the end of the experiments of Section 4.3.2, for each app.

4.3.1 Simulated apps

We compare our agent against the baseline randomly clicking agent on three separate simulated apps. Each simulation is designed so that the agent receives a reward signal with value 10, interpreted as method coverage, whenever a never before visited screen is reached. Each app is restarted after 200 consecutive actions from the agent, but the agent gets to keep its internal state and knowledge, allowing it to explore states from the beginning.

The first app (Figure 4.5) is a series of distinct screens, each with two buttons in identical

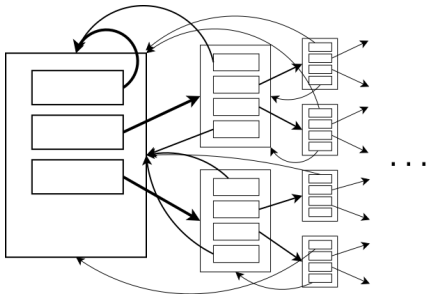


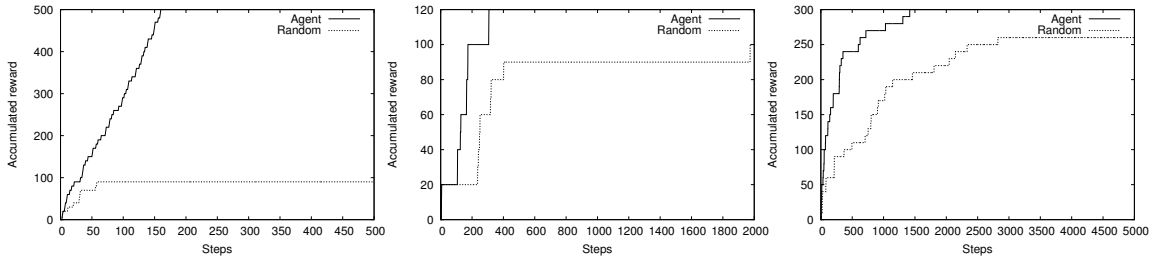
Figure 4.7: Simulated App 3: Binary tree

positions. The screens are ordered in series, such that clicking on the first button of each screen transitions the app to the next screen, and the second button takes the app to the previous screen. The first screen has no second (back) button, whereas the last has no button in the first (forward) position. Figure 4.8a plots the performance of our agent against the random agent on an instance of this simulated app with 50 distinct screens. We note that random exploration quickly gets stuck, and over 500 actions it spends all of its time exploring the first 10 screens. This limitation is reached even before the simulated app is reset to the initial screen at the 200 actions mark. The reason is that, in every screen, besides a high chance of clicking on the background, the random agent has exactly the same chance of hitting the next or the back button. Although our proposed agent has no notion of which screen it is in, it will quickly recognize and avoid the background and, given the choice, will prefer clicking on a button it hasn't clicked on before, rather than one it has. This is enough to allow it to greatly outperform random in this synthetic app.

The second simulated app (Figure 4.6) is composed of a series of pairs of screens. Each pair is composed of two screens with eight buttons each, a click on each button takes the app to the opposite screen of the pair, except from a single button on the second screen of the pair (selected at random) which, when clicked, takes the simulated app to the next pair of screens. We built a simulation formed by six such pairs of screens, and compared the performance of our agent against the random agent on them. Figure 4.8b plots the results. Once again, our agent efficiently explores the app, trying each button and finding all the screens within the first 400 actions. The random agent, on the other hand, has not yet found the sixth pair after 2000 actions.

Finally, Figure 4.7 depicts the third simulated application: A binary tree of screens where each screen has one button going directly to the root of the tree, one button each for each of the two screens along each branch of the tree, and a final button that goes to the parent, or previous, screen. Figure 4.8c plots the performance of our agent against the random agent over 5000 events on this simulated app. Although this simulation is slightly tougher than the last two for our agent, it still manages to significantly outperform random clicking.

We show these results on artificial navigation challenges to put into context the more challenging task of exploring real applications, and to show that our agent is able to explore relatively complex



(a) S.App 1: 50 Back/Next **(b) S.App 2: 6 Tangled pairs** **(c) S.App 3: Binary tree (d=5)**

Figure 4.8: Agent vs Random on simulated apps

flows without any explicit model of the apps' navigational structure.

4.3.2 Google Play apps

We now describe the results of comparing our agent against the random agent on 15 real world applications taken from the Google Play Store.

To select this dataset of popular real world Android applications, on May 8, 2016, we scraped the Play store's top 10 free apps lists for each of the following 22 different application categories: books & references, business, comics, communication, education, entertainment, finance, health & fitness, lifestyle, media & video, medical, music & audio, news & magazines, photography, productivity, shopping, social, sports, tools, transportation, travel & local, and weather. These were are all the Android app categories listed at the time in the Play store, excluding games, wallpapers and Android Wear applications. We omitted the later three categories from consideration, since they have markedly different UI and user interaction patterns from other applications and are thus extreme outliers for our approach. This gave us a list of 220 applications in total. We downloaded the binaries for those 220 apps from the Play store between May 8, 2016 and Jan 17, 2017.

Due to limitations of the `Ella` tool, many of those binary apps (.apk files) could not be properly instrumented to provide real time method coverage feedback, which we use as a metric in our evaluation, and had to be discarded. This is not a limitation on our agent, which can exercise any app which we can run in our emulator as long as we can take screenshots from it and send back GUI actions. It is instead a limitation of our evaluation methodology and the tools we use to gather coverage information as part of that evaluation. Additionally, we had trouble downloading a few apps from the Play store[2], and a few of the apk's we did manage to download and instrument would not run on our Android emulator or would crash upon launch. Out of the original 220 apps: 39 (18%) failed to download, 126 (57%) failed to instrument and 9 (4%) failed to run (instrumented or

[2]The Play store, among other things, tries to limit downloads of apps that are not compatible with the devices registered to a particular user's Google account

Table 4.1: Dataset of Apps

#	Package name	Category	Rank	sha1sum
1	com.indeed.android.jobsearch	business	1st	bf1e0f91ff5837c774d02cad4ab3a7b645e58569
2	com.duolingo.app	education	1st	06668905e29b98c970ad2f9721e5184a59423cb0
3	com.creditkarma.mobile	finance	1st	ac997d182a1ea43dabc634d30097c0d26ee67c8a
4	com.goodrx	medical	1st	4b26cbaeb371cd280e275e77f8c552aeab67796e
5	com.surpax.ledflashlight.panel	productivity	1st	93cc38d0f37f06e3aebc94e6cd9b0a829b098cd6
6	com.snapchat.android	social	1st	ca8bc8ea770ccb8e7cb4d80d6bbbc46974ef066f
7	com.yelp.android	travel&local	1st	900d23bf94c2e286a00fba587ef5836157e95cf6
8	com.marvel.comics	comics	2nd	0ca52ea44026a495569d361a9a65c01c7cc4822e
9	com.adpmobile.android	business	3rd	4cd44d2bb9c5f9631ce40f9258fb45dd5fd46f80
10	com.instagram.android	social	3rd	60c795170a6aefc98f8bf0cc0d3de1317bc64ab5
11	com.duapps.cleaner	tools	3rd	2e8b01c90a610b9c231f7228f57e9c91cbb4b56d
12	com.squareup	business	4th	67053bfb213f5d5c93bdfc5ddfc3b49d946c2563
13	com.happy2.bbmanga	comics	4th	e4137b99b83771591e1d7497f7b8fc5ad538da7f
14	com.wf.wellsfargomobile	finance	4th	8a439b8177ae60880b489af956dabb2a3df1a740
15	com.cvs.launchers.cvs	health&fitness	4th	c8839592ff9fcbff83e7b66f2340e17967e53703

not), leaving 46 (21%) for our experiments.

Due to time and resource limitations, we restricted ourselves to the topmost 15 of those remaining apps, defined as follows: We went over the original 220 apps, looking first at the top app from each category, followed by the top two app from each category, and so on and so forth, in a round-robin fashion, until we had 15 apps which could be downloaded, instrumented and would run without issue. Table 4.1 lists those apps by package name, category, rank within the category and SHA1 hash of the .apk file for the specific version we tested. The experiments described in this section and the next were performed on this final set.

Figure 4.9 shows the coverage achieved by running both our agent and the random agent on each of the 15 apps for 5000 actions, averaged over four runs for each condition. The x-axis represents the cumulative number of actions (taps on the screen) performed by each agent. The y-axis represents the cumulative method coverage (the number of distinct methods executed) at that point, with the left y-axis showing the total number, and the right y-axis showing it as a percentage of the total number of methods inside the app's APK file. Note that the method coverage shown in the figures does not start at 0, since some portion of the code for each app is executed at installation or when starting up the application's initial screen, before any action by the agent.

We use method coverage since Google Play applications do not have enough debug information to recover line coverage metrics. Please note that since APK files include all third-party libraries used by the app, other than the standard Android libraries, these apps can contain large portions of unreachable code which skews the coverage percentage metric to be fairly low, even if we could achieve perfect exploration of the application behavior. An additional source of uncovered code is due to some of the apps having significant functionally behind a log-in or sign-up screen, or requiring complex set up to enable some of their functionality. However, this limitation applies equally to our agent and the random agent in their present form, allowing for a fair comparison.

For both our agent and the random agent, we had the app under test reset after every 200 actions

of the agent. To ensure that the app properly reset to the same initial state each time, we didn't merely restart the app, but fully restored the emulator to a system snapshot taken before the app was installed, and then installed the corresponding APK again, after every 200 action "episode". Additionally, if the app or emulator crashed, for any reason, we reset the app before the agent's next observation. In both cases, the internal state of our agent was not cleared or reset in any way, only the state of the app under test.

To control for the agent's randomness and application non-determinism, we ran each condition four times for each app, all of the numbers and plots in this section and the next are from taking the average of those four runs in each case.

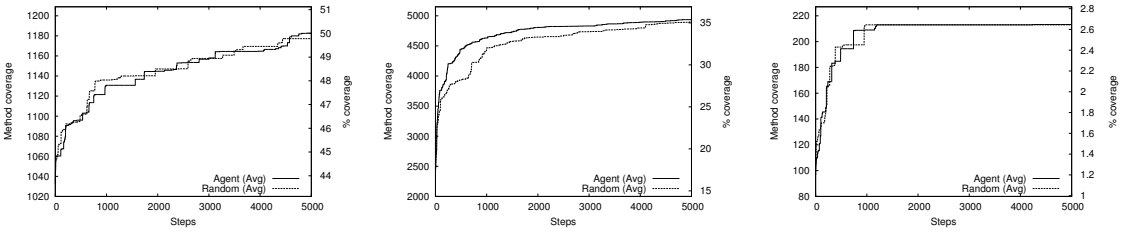
We can see in Figure 4.9, that our agent clearly outperforms random in 9 out of 15 apps (4.9b, 4.9e, 4.9f, 4.9g, 4.9h, 4.9i, 4.9m, 4.9n and 4.9o), either in total method coverage achieved or the number of actions required to reach such coverage. For no app does our agent under-perform with respect to the random agent. Additionally, a few of the apps (4.9c, 4.9d and 4.9l) are such that either agent usually achieves close to full coverage during the first fifth of our runs, in which case our agent has not had much time to learn from previously seen screenshots.

Of the remaining three apps, they each exhibit one or two of the following three properties, which make it harder for our agent to learn to explore them. First, 4.9a and 4.9j quickly run into the log-in or sign-up screen wall, meaning most of the exploration has to go around the main flow (e.g. by testing the password reset feature, or by finding menu options that are available without log-in). Second, the app might have particularly cluttered screens, where our agent's flood fill technique is rendered useless by large groups of visually irregular widgets packed together with almost no background squares. App 4.9a exhibits this problem due to a large number of forms, which are also hard to explore in general, while app 4.9k runs into this issue due to a substantial number of embedded ads taking over otherwise unused screen real state. Finally, app 4.9j has a dynamically changing background which, on the main log-in screen, takes over 70% of the screen, breaking our heuristic for detecting constantly changing portions of the screen. Figure 4.10 shows this screen at three different times, note that not only the color of the background is different every time, but it is also a gradient background, which our current implementation of flood fill does not handle well.

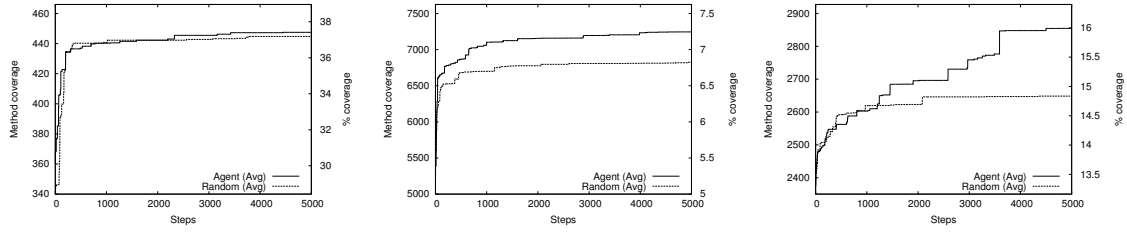
In general, the following improvements should help our agent deal with cases like the ones we just discussed: improve our similarity metric for flood fill to detect visually similar areas that are not made of identical single color squares, and allow our agent to select higher-level actions in addition to taps, such as "type email", to get around sign up forms.

4.3.3 Memory depth

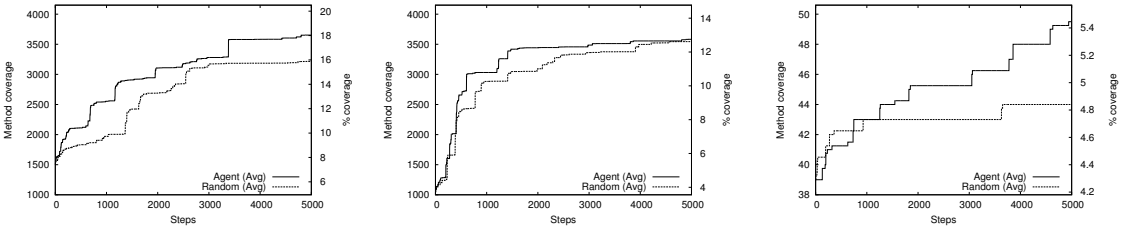
Another interesting dimension on which to evaluate our approach is the depth and total size of the agent's memory. As described back in Section 4.1 and shown in Figure 4.1, our agent keeps a memory of seen image patches as a map associating each position in the grid of squares defining



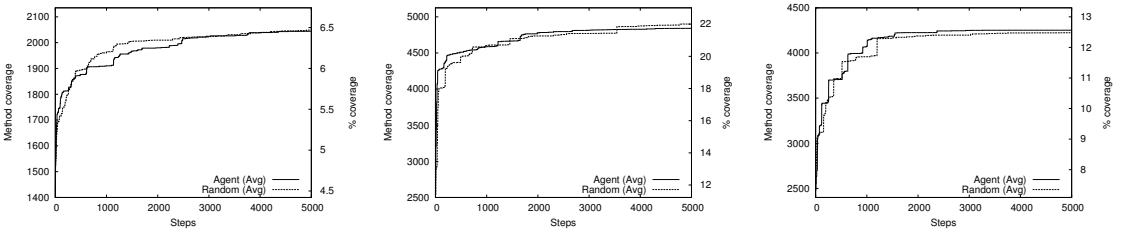
(a) com.indeed.android.jobsearch (b) com.duolingo.app (c) com.creditkarma.mobile



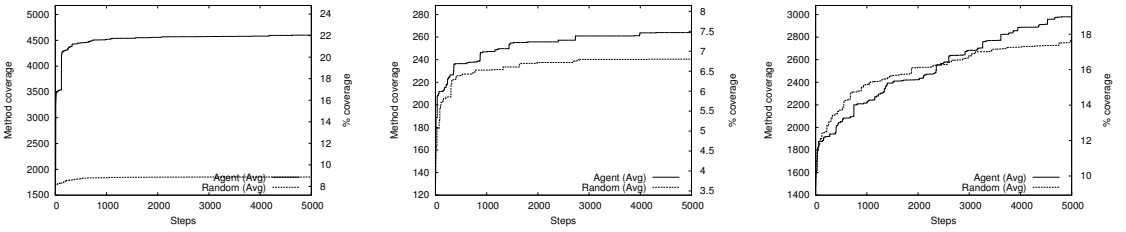
(d) com.goodrx (e) com.surpax.ledflashlight (f) com.snapchat.android



(g) com.yelp.android (h) com.marvel.comics (i) com.adpmobile.android



(j) com.instagram.android (k) com.duapps.cleaner (l) com.squareup



(m) com.happy2.bbmanga (n) com.wf.wellsfargomobile (o) com.cvs.launchers.cvs

Figure 4.9: Method coverage over time for each of our apps under test

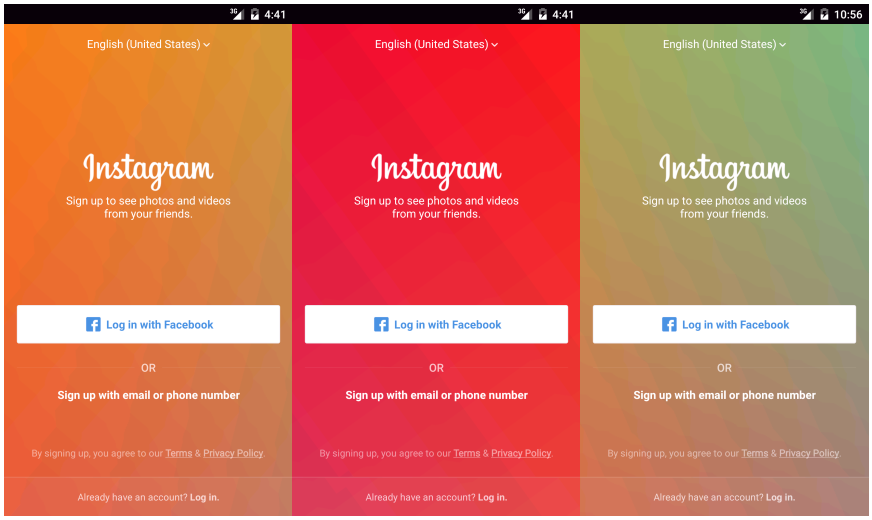


Figure 4.10: com.instagram.android's always changing background

available actions with a list of pairs. Each pair matches a seen image patch with a value p indicating how likely the agent thinks that such a patch in that grid location will be clickable in the app. Figure 4.2 gives the depth and size of this data structure by the end of the 5000 actions for each Google Play app in our experiments. As in the previous section, all the numbers are averages over four runs, explaining the fractional numbers of pairs.

The depth number is interesting, as it reflects the maximum number of distinct image patches found by the agent for a single grid location at the end of the run. Given that this number is sometimes over 100 different image patches, and most apps do not have a hundred different underlying activities, this is additional evidence that a similarity metric other than equality would work better to let the agent know whether or not it should treat a previously unseen patch as clickable, given the p calculated for other "similar" patches in that same grid location. Such an extension is left for future work.

Additionally, the total size of the agent memory is significant as a measure of its resource consumption. We find that, when storing only a hash of the image patch in question as an integer value and the float value p, the size of the memory in bytes is actually fairly small, on the order of tens or hundreds of KB for most apps, even after sampling 5000 screenshots. At that size, the memory usage of our agent is actually dominated by the space required to process the current screenshot for each cycle, rather than the persistent agent memory. Some of our earliest versions of the agent used to store an actual representation of the image patch in question rather than a hash, requiring about 7,504 bytes per pair in the agent memory. For `com.indeed.android.jobsearch`, for example, this would have meant that the agent would use $>$ 142 MB of RAM to store the memory data structures.

The numbers above indicate that, using hashes to encode the image patches seen previously,

Table 4.2: Depth and Total Size of the Agent's Memory

#	Package name	Memory depth (in pairs)	Memory size (in pairs)	Memory size (in KB)
1	com.indeed.android.jobsearch	43.5	18962.25	151.7 KB
2	com.duolingo.app	89	18721.25	149.8 KB
3	com.creditkarma.mobile	93.25	4324.5	34.6 KB
4	com.goodrx	32.5	15047.75	120.4 KB
5	com.surpax.ledflashlight.panel	13.5	1866.75	14.9 KB
6	com.snapchat.android	136.75	8327	66.6 KB
7	com.yelp.android	80	12770.75	102.1 KB
8	com.marvel.comics	61.75	6143.25	49.1 KB
9	com.adpmobile.android	22	8699.5	69.6 KB
10	com.instagram.android	41.5	13831.75	110.7 KB
11	com.duapps.cleaner	41	12657	101.3 KB
12	com.squareup	128	5367.25	42.9 KB
13	com.happy2.bbmanga	63.25	9574.5	76.6 KB
14	com.wf.wellsfargomobile	40.5	10211.75	81.7 KB
15	com.cvs.launchers.cvs	80682	14562	116.5 KB
Average		5437.9	10737.8	85.9 KB
Median		61.75	10211.75	81.7 KB

our agent is quite reasonable in terms of memory requirements. This is true even over a long app exploration session. Thus, it is practical to consider our agent as a replacement for Android monkey or other fully-random exploration agents, which use essentially constant memory.

Chapter 5

Related Work

This chapter briefly describes some of the most significant work related to the problems and techniques explored in the previous chapters.

We began the present work by describing the problem of obtaining platform specifications to be consumed by a whole-program static analysis system targeting applications embedded in a complex framework. Section 5.1 briefly describes other such static analysis systems, their use of specifications, and a complementary technique to our own for specification mining.

In Chapter 2, we presented a technique to automatically mine such specifications using dynamic analysis. Section 5.2 explores other approaches to dynamic specification mining. Section 5.3 gives a brief overview of dynamic taint tracking analysis techniques that are similar to the base taint tracking technique used in our specification mining. Section 5.4 notes a few tools which can be used for tracing executions, similar to DroidRecord.

Two other contributions of this work, described in Chapters 3 and 4, are forms of automated app exploration and testing. Section 5.5 describes alternative approaches to mobile GUI testing and gives an overview of that area. Section 5.6 mentions relevant work related to Delta Debugging and MDPs, both of which we used as part of the technique for minimizing random GUI event traces described in Chapter 3.

5.1 Whole-program static taint analysis and its specifications

A number of techniques and tools [30, 54, 81, 70, 105] have been developed for whole-program taint analysis. See [94] for a survey of work in this field. For applications that run inside complex application frameworks these analyses often must include some knowledge of the framework itself. F4F [102] is a scheme for encoding framework-specific knowledge in a way that can be processed by a general static analysis. In F4F, any models for framework methods must be written manually. In contrast, we are most interested in techniques that produce framework models automatically, with

minimal human effort. The trade-off is that, for the time being, F4F models, and those of other general manual framework modeling languages, are potentially more expressive than the kinds of specifications we are able to mine using Modelgen.

Flowdroid [8] is a context-, flow- and object-sensitive static taint analysis system for Android applications, which can analyze Android platform code directly. By default, it uses models or 'shortcuts' for a few platform methods as a performance optimization and to deal with hard-to-analyze code. Flowdroid's shortcuts are also information-flow specifications of a slightly more restrictive form than that used by Modelgen. Thus, it seems likely the FlowDroid shortcuts could also be mined successfully from tests.

The technique presented in [14] uses static analysis to infer specifications of framework methods such that those specifications complete information flow paths from sources to sinks. Since this technique does not analyze the code of the framework methods, it can and does suggest spurious models, which must be filtered by a human analyst. This technique is complementary to the one we described in Chapter 2; DroidRecord can be used to validate models inferred by this technique. There has also been some previous work on identifying sources and sinks in the Android platform based on the information implicitly provided by permission checks inside API code [36, 9, 12] or by applying machine learning to some of the method's static features [90]. This work could be combined with our method for inferring specifications to enable fully automatic explicit information flow analysis (i.e., with no manual annotations).

5.2 Dynamic techniques for creating API specifications

Many schemes have been proposed for extracting different kinds of specifications of API methods or classes from traces of concrete executions. Closest to the work presented in this dissertation is work on producing dynamic dependence summaries of methods as a way to improve the performance of whole-program dynamic dependence analysis [85]. Dependence analysis of some form is a prerequisite for explicit information flow analysis, since it involves determining which program values at which point in the execution are used to compute every new program value. Although our use case, and thus evaluation, is very different than that of [85], the specifications produced are somewhat related. In [85], a specification that a flows to b means literally that a location named by a is used to compute a value in a location named by b. In our framework, a specification that a flows to b means that some value reachable from a is used to compute some value reachable from b. Thus, the major difference is that we "lift" the heap-location level flows to abstract flows between method arguments and between arguments and the return value of the method, as described in 2.4.3. This lifting step requires additional infrastructure to maintain colors in the dynamic analysis, an issue that does not arise in dynamic dependence analysis. The added abstraction reduces the size of the summaries and allows us to generalize from fewer traces, though with a potential loss in precision, a trade-off which

our results suggest is justified.

Using dynamic analysis to compute specifications consumed by static analysis has also been heavily explored. However, most such specifications focus on describing control-flow related properties of the code being modeled. A large body of work (e.g. [16, 4, 108, 71, 114, 112, 27, 41, 73, 72, 66]) constructs Finite State Automata (FSA) encoding transitions between abstract program states.

Other approaches focus on inferring program invariants from dynamic executions, such as method pre- and post-conditions (Daikon [84, 34, 35]), array invariants [83] and algebraic "axioms" [55]. Data from dynamic executions has also been used to guide sound automated equivalence checking of loops [100]. Another relevant work infers static types for Ruby programs based on the observed run-time types over multiple executions [57]. Finally, program synthesis techniques have been used to construct simplified versions of API methods that agree with a set of given traces on their input and output pairs [89].

5.3 Dynamic taint tracking and related analyses

Dynamic taint tracking uses instrumentation and run-time monitoring to observe or confine the information flow of an application. Many schemes have been proposed for dynamic taint tracking [51, 26, 31, 10]. An exploration of the design space for such schemes appears in [96]. Dytan [26] is a generic framework capable of expressing various types of dynamic taint analyses. Our technique for modeling API methods is similar to dynamic taint tracking, and could in principle be reformulated to target Dytan or some similar general dynamic taint tracking framework. However, heap-reachability and all of our analysis would have to be performed online, as the program runs, which might exacerbate timing dependent issues with the Android platform.

As mentioned previously, dependence analysis is also related to information flow analysis, and the large body of work in dynamic dependence analysis is therefore also relevant to our own (e.g. [104, 56] and references therein).

5.4 Tools for tracing dynamic executions

Query languages such as PTQL [43] and PQL [76] can be used to formulate questions about program executions in a high-level DSL, while tools like JavaMaC [63], Tracematches [1], Hawk [28] and JavaMOP [59] permit using automata and formal logics for the same purpose. Frameworks like RoadRunner [39] and Sofya [64] allow analyses to subscribe to a stream of events representing the program execution as it runs.

5.5 Automated GUI testing tools for mobile applications

Many tools exist for generating GUI event traces to drive Android apps. These tools, sometimes collectively called 'monkeys', are commonly used to automatically generate coverage of an app's behavior or to drive app execution as part of a dynamic analysis system.

Dynodroid [74] improves on the standard Android Monkey by automatically detecting when the application registers for system events and triggering those events as well as standard GUI events. It also provides multiple event generation strategies which take the app context into account and it allows the user to manually provide inputs when exploration is stalled (e.g. at a login screen). Tools such as GUIRipper [2]/ MobiGUITAR [3], AppsPlayground [91], ORBIT [115], SwiftHand [21], and A^3E [11] dynamically crawl each app while building a model which records observed states, allowed events in each state, and state transitions. The generated model is used to systematically explore the app. PUMA [53] provides a general framework over which different model-based GUI exploration strategies can be implemented. ACTEve [6] is a concolic-testing framework for Android, which symbolically tracks events from the point in the framework where they are generated, up to the point at which the app handles them. EvoDroid [75] generates Android input events using evolutionary algorithms, with a fitness function designed to maximize coverage. Brahmastra [15] is another tool for driving the execution of Android apps, which uses static analysis and app rewriting to reach specific components deep within the app. Application rewriting can bypass many difficulties related to GUI exploration, but may produce execution traces which are not truly possible in the unmodified app.

A recent survey paper by Choudhary et al. [22] compares many of the tools above and seems to suggest that the standard Android Monkey is competitive in coverage achieved in a limited time. One explanation is that Monkey compensates for what it lacks in sophistication by maintaining a higher rate of event generation. Of course, this result could also be the effect of comparing research prototypes against an industry standard tool, which performs more robustly even while using less sophisticated techniques. A third explanation is that Monkey's robustness, and therefore efficacy, comes from the fact that it treats apps under test in a strong black box manner, and is thus less sensitive to implementation details of the application (e.g. programing language, UI framework or server vs client side code partitioning) compared to the more sophisticated tools. In either case, the effectiveness of the standard Monkey to achieve high coverage, along with the relative noisiness of the traces produced, justifies our focus on trace minimization.

In addition to general test input generation tools for Android, many dynamic analysis tools include components that drive exploration of the app being analyzed (e.g. [67, 82, 69, 92]). Input fuzzers have been used to generate some types of application inputs for mobile apps, such as inter-app communication messages [95] or structured input to test for invalid data handling [116]. To our knowledge, fuzzers haven't been explored in the context of generating GUI event traces or GUI exploration in general.

In addition to minimizing traces generated by Monkey-style tools, our approach is also applicable to minimizing recorded user-interaction traces. Tools such as RERAN [44], Mosaic [52] and VALERA [58] could be used to record the actions of a human tester as an input event trace for our method.

Besides automated input generation tools, GUI-aware tests for Android applications tend to be encoded as testing scripts in frameworks such as Selendroid [99], Robotium [93], Calabash [111] or Espresso [48]. These frameworks allow scripting specific interaction scenarios with an Android app and adding checks to generate an effective application test suite. A promising line of future work is to automatically transform automatically-generated and minimized execution traces into test scripts expressed in any of these frameworks, as a way to provide a starting point for test suite writers.

5.6 Delta debugging and MDPs

The core of our event trace minimization algorithm from Chapter 3 is based on delta debugging. Delta debugging is a family of algorithms for sequence minimization and fault isolation, described originally by Zeller et al. [117, 119]. The technique has been extended to many scenarios [118, 20, 80, 17].

The work by Scott et al. [98, 97], extends delta debugging to minimize execution traces which trigger bugs within non-deterministic distributed systems. They run standard delta debugging over the traces of external (user triggerable) events. Then, to check each subtrace of external events, they instrument the system under test and explore the space of possible interleavings of internal events. By contrast, we treat non-deterministic Android applications in a black box manner and rely on modeling the probability of success of traces of external events, independently of the internal workings of the system being tested. For the specific case of minimizing GUI event traces in Android applications, an important source of non-determinism turns out to be responses from network services outside our control, justifying the need for a blackbox approach. In other scenarios, the tradeoff between both approaches likely depends on the complexity of the internals of the system being tested and the 'success' probability of the original trace to be minimized.

A Markov Decision Process is a standard model within the reinforcement learning literature (see e.g. [60, 103]). MDPs are used to solve a variety of problems across multiple domains, including optimizing resource consumption in mobile phones [19]. To the best of our knowledge, we are the first to apply MDPs to the problem of trace minimization in testing.

Chapter 6

Conclusion

Mobile apps, like many modern applications, exist embedded inside a large and complex platform. Static analysis of these apps then entails analysis or modeling of the platform. Since these platforms often pose significant challenges to whole-program static analysis techniques, the usual approach is to provide specifications of the platform methods, which depend on the type of static analysis being performed. We presented a technique for mining a particular class of specifications, those to be consumed by a whole-program static explicit information flow analysis system, using dynamic analysis techniques over a set of collected concrete program executions. We instantiated this technique in Modelgen, a tool targeting the Android platform and a concrete existing static analysis system (STAMP). Modelgen specifications are highly precise and provide high recall with respect to our existing manual models. They also allow our static analysis to find true flows it misses despite years of manual model construction effort. Furthermore, such specifications can be inferred from a relatively small set of execution traces.

We propose that similar techniques, based on dynamic analysis, can be used to construct different types of specifications for other kinds of static analysis clients. As we mention in Chapter 5, such techniques exist for mining specifications as diverse as: source and sink annotations, method pre- and post- conditions, and FSA-based encodings of program state transitions. We observe that automatic mining of specifications from concrete executions is an effective way to significantly reduce the manual effort involved in modeling the platform inside which the apps to be analyzed run, and that we expect the types of specifications covered by similar approaches to increase over time.

At the same time, producing the executions that trigger the full range of behaviors required to infer useful specifications requires either a comprehensive test suite or a way to systematically explore the app or platform behavior. If the corresponding test cases or executions must be manually generated, then we have merely transfered the required human effort from writing specifications to writing tests. The manually written test cases, for example, act as an implicit specification of the platform behaviors. We thus also explored the problem of automated testing of mobile applications.

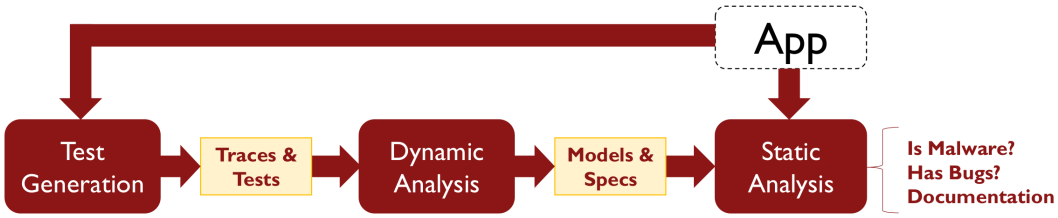


Figure 6.1: Combining specification mining and automated testing

First, we presented a technique for minimizing random GUI event traces, generated by tools such as Monkey. As mentioned in Chapter 5, this is a useful technique, since others have observed that Monkey performs surprisingly well in practice in terms of the coverage obtained and its robustness to new real-world apps. Our GUI event trace minimization technique in particular is based on a delta debugging extension. It handles non-determinism, which we have shown is a pervasive issue in app behavior. We have also presented two strategies for efficient trace selection: a custom heuristic and one based on modeling the problem as an MDP. Evaluation of our trace minimization method shows that the resulting traces are 100x smaller while still reaching the same activity with high probability.

Second, we presented a new tool for automated app exploration, which preserves Monkey's robustness by treating the app's code as a black box and reacting exclusively to changes being shown on the screen in response to the tool's actions. We show that this tool outperforms random testing for many real world apps, and doesn't seem to be at a disadvantage for any particular app. Although our method for evaluating the effectiveness of this agent required app instrumentation, preventing us from including some real world apps in our evaluation, the tool itself does not need any changes to the APKs it takes as input and is insensitive to details of the app internals, such as programming language, UI toolkit, webview/OpenGL/Native view rendering, or our ability to instrument or analyze the code.

It is important to note that automated testing methods in general, and our techniques in particular, aim to generate relevant executions or test cases using only the executable application itself as input. This means that, in combination with specification mining techniques that take concrete executions as input, we can significantly reduce, and might be able to eventually eliminate, the manual effort involved in modeling the platform as part of a whole-program static analysis system.

We believe future work on both mining new and more expressive kinds of specifications, as well as on faster and more comprehensive automated testing, will eventually lead us to systems like the one pictured in Figure 6.1. Here, the application itself is fed to an automated testing module, which generates the required test cases or execution traces to drive a specification mining module. This later module generates specifications of the platform or environment on which the application runs, which are then fed to a static analysis system, together with the original application, which

produces some useful information about the application. This information can be used for detecting potentially malicious behavior, as STAMP does, bug finding, or other tasks such as automatically generating documentation about the application's behavior.

Bibliography

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA*, pages 345–364, 2005.

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI ripping for automated testing of Android applications. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pages 258–261, 2012.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.

[4] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.

[5] Saswat Anand. ELLA: A tool for binary instrumentation of android apps - https://github.com/saswatanand/ella.

[6] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 59, 2012.

[7] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[8] 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. In *PLDI*, pages 259–269, 2014.

[9] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android permission specification. In *CCS*, pages 217–228, 2012.

[10] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *POPL*, pages 165–178, 2012.

[11] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 641–660, 2013.

[12] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing Android. *IEEE Transactions on Software Engineering*, 40(6):617–632, 2014.

[13] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *SOAP*, pages 27–38, 2012.

[14] Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In *POPL*, pages 553–566, 2015.

[15] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 1021–1036, 2014.

[16] Alan W Biermann and Jerome A Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 100(6):592–597, 1972.

[17] Martin Burger and Andreas Zeller. Minimizing reproduction of software failures. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 221–231, 2011.

[18] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.

[19] Tang Lung Cheung, Kari Okamoto, Frank Maker III, Xin Liu, and Venkatesh Akella. Markov decision process (MDP) framework for optimizing software on mobile phones. In *Proceedings of the 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009, Grenoble, France, October 12-16, 2009*, pages 11–20, 2009.

[20] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *Proceedings of the 11th International Symposium on Software Testing and Analysis, ISSTA 2002, Rome, Italy, July 22-24, 2002*, pages 210–220, 2002.

[21] Wontae Choi, George C. Necula, and Koushik Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 623–640, 2013.

[22] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for Android: Are we there yet? In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 429–440, 2015.

[23] Lazaro Clapp. Droidrecord: Android instrumentation recording and specification mining toolset - https://bitbucket.org/lazaro_clapp/droidrecord.

[24] Lazaro Clapp, Saswat Anand, and Alex Aiken. Modelgen: mining explicit information flow specifications from concrete executions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 129–140, 2015.

[25] Lazaro Clapp, Osbert Bastani, Saswat Anand, and Alex Aiken. Minimizing GUI event traces. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 422–434, 2016.

[26] James A. Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA*, pages 196–206, 2007.

[27] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with ADABU. In *WODA*, pages 17–24, 2006.

[28] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of Java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[29] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings*, pages 77–101, 1995.

[30] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[31] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.

[32] William Enck, Machigar Ongtang, and Patrick D. McDaniel. Understanding android security. *IEEE Security & Privacy*, 7(1):50–57, 2009.

[33] Michael D. Ernst. Invited talk static and dynamic analysis: synergy and duality. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04, Washington, DC, USA, June 7-8, 2004*, page 35, 2004.

[34] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[35] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, pages 143–160, 2010.

[36] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 627–638, 2011.

[37] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: user attention, comprehension, and behavior. In *Symposium On Usable Privacy and Security, SOUPS '12, Washington, DC, USA - July 11 - 13, 2012*, page 3, 2012.

[38] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *FSE*, pages 576–587, 2014.

[39] Cormac Flanagan and Stephen N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *PASTE*, pages 1–8, 2010.

[40] The Apache Software Foundation. Apache struts - http://struts.apache.org.

[41] Mark Gabel and Zhendong Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *SIGSOFT*, pages 339–349, 2008.

[42] Raffaele Garofalo. *Building enterprise applications with Windows Presentation Foundation and the model view ViewModel Pattern*. Microsoft Press, 2011.

[43] Simon Goldsmith, Robert O'Callahan, and Alexander Aiken. Relational queries over program traces. In *OOPSLA*, pages 385–402, 2005.

[44] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd D. Millstein. RERAN: timing- and touch-sensitive record and replay for Android. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 72–81, 2013.

[45] Google. Android documentation: Working with system permissions - https://developer.android.com/training/permissions/index.html.

[46] Google. Android runtime (ART) and dalvik documentation - https://source.android.com/devices/tech/dalvik/.

[47] Google. Compatibility test suite (Android) - https://source.Android.com/compatibility/cts-intro.html.

[48] Google. Espresso - https://developer.android.com/training/testing/ui-testing/espresso-testing.html.

[49] Google. Keeping your app responsive - https://developer.Android.com/training/articles/perf-anr.html.

[50] Google. UI/Application exerciser monkey - https://developer.android.com/tools/help/monkey.html.

[51] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *ACSAC*, pages 303–311, 2005.

[52] Matthew Halpern, Yuhao Zhu, Ramesh Peri, and Vijay Janapa Reddi. Mosaic: cross-platform user-interaction record and replay for the fragmented Android ecosystem. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*, pages 215–224, 2015.

[53] Shuai Hao, Bin Liu, Suman Nath, William G. J. Halfond, and Ramesh Govindan. PUMA: programmable ui-automation for large-scale dynamic analysis of mobile apps. In *The 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'14, Bretton Woods, NH, USA, June 16-19, 2014*, pages 204–217, 2014.

[54] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *POPL*, pages 365–377, 1998.

[55] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. Discovering documentation for Java container classes. *IEEE Transactions on Software Engineering*, 33(8):526–543, 2007.

[56] Justin Holewinski, Ragavendar Ramamurthi, Mahesh Ravishankar, Naznin Fauzia, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. Dynamic trace-based analysis of vectorization potential of applications. In *PLDI*, pages 371–382, 2012.

[57] Jong hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for Ruby. In *POPL*, pages 459–472, 2011.

[58] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. Versatile yet lightweight record-and-replay for Android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 349–366, 2015.

[59] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. JavaMOP: Efficient parametric runtime monitoring framework. In *ICSE*, pages 1427–1430, 2012.

[60] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *J. Artif. Intell. Res. (JAIR)*, 4:237–285, 1996.

[61] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 423–434, 2013.

[62] Patrick Gage Kelley, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman M. Sadeh, and David Wetherall. A conundrum of permissions: Installing applications on an android smartphone. In *Financial Cryptography and Data Security - FC 2012 Workshops, USEC and WECSR 2012, Kralendijk, Bonaire, March 2, 2012, Revised Selected Papers*, pages 68–79, 2012.

[63] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Java-MaC: A run-time assurance tool for Java programs. *Electronic Notes in Theoretical Computer Science*, 55(2):218–235, 2001.

[64] Alex Kinneer, Matthew B. Dwyer, and Gregg Rothermel. Sofya: Supporting rapid development of dynamic program analyses for Java. In *ICSE Companion*, pages 51–52, 2007.

[65] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[66] Ivo Krka, Yuriy Brun, Daniel Popescu, Joshua Garcia, and Nenad Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *ICSE*, pages 179–182, 2010.

[67] Kyungmin Lee, Jason Flinn, Thomas J. Giuli, Brian Noble, and Christopher Peplin. AMC: verifying user interface properties for vehicular applications. In *The 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'13, Taipei, Taiwan, June 25-28, 2013*, pages 1–12, 2013.

[68] Avraham Leff and James T Rayfield. Web-application development using the model/view/-controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*, pages 118–127. IEEE, 2001.

[69] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. DECAF: detecting and characterizing ad fraud in mobile apps. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 57–70, 2014.

[70] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.

[71] David Lo and Siau-Cheng Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *FSE*, pages 265–275, 2006.

[72] David Lo, Leonardo Mariani, and Mauro Pezzè. Automatic steering of behavioral model inference. In *ESEC/FSE*, pages 345–354, 2009.

[73] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *ICSE*, pages 501–510, 2008.

[74] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: an input generation system for Android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 224–234, 2013.

[75] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 599–609, 2014.

[76] Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*, pages 365–383, 2005.

[77] Lucas Matney. Google has 2 billion users on android, 500m on google photos, May 2017. [Online; posted 17-May-2017].

[78] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-cfa paradox. *CoRR*, abs/1311.4231, 2013.

[79] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[80] Ghassan Misherghi and Zhendong Su. HDD: hierarchical delta debugging. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 142–151, 2006.

[81] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.

[82] Suman Nath, Felix Xiaozhu Lin, Lenin Ravindranath, and Jitendra Padhye. Smartads: bringing contextual ads to mobile apps. In *The 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'13, Taipei, Taiwan, June 25-28, 2013*, pages 111–124, 2013.

[83] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using dynamic analysis to discover polynomial and array invariants. In *ICSE*, pages 683–693, 2012.

[84] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA*, pages 229–239, 2002.

[85] Vijay Krishna Palepu, Guoqing (Harry) Xu, and James A. Jones. Improving efficiency of dynamic analysis with dynamic dependence summaries. In *ASE*, pages 59–69, 2013.

[86] Hao Peng, Christopher S. Gates, Bhaskar Pratim Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using probabilistic generative models for ranking risks of android apps. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 241–252, 2012.

[87] S Pichai. Google I/O 2017-keynote [video. 6: 30m], 2017.

[88] Mike Potel. Mvp: Model-view-presenter the taligent programming model for c++ and java. *Taligent Inc*, page 20, 1996.

[89] Dawei Qi, William N. Sumner, Feng Qin, Mai Zheng, Xiangyu Zhang, and Abhik Roychoudhury. Modeling software execution environment. In *WCRE*, pages 415–424, 2012.

[90] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing Android sources and sinks. In *NDSS*, 2014.

[91] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: automatic security analysis of smartphone applications. In *Third ACM Conference on Data and Application Security and Privacy, CODASPY'13, San Antonio, TX, USA, February 18-20, 2013*, pages 209–220, 2013.

[92] Lenin Ravindranath, Suman Nath, Jitendra Padhye, and Hari Balakrishnan. Automatic and scalable fault detection for mobile applications. In *The 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'14, Bretton Woods, NH, USA, June 16-19, 2014*, pages 190–203, 2014.

[93] Robotium. Robotium - https://github.com/robotiumtech/robotium.

[94] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[95] Raimondas Sasnauskas and John Regehr. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA), WODA+PERTEA 2014, San Jose, CA, USA, July 22, 2014*, pages 1–5, 2014.

[96] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *SSP*, pages 317–331, 2010.

[97] Colin Scott, Aurojit Panda, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. Minimizing faulty executions of distributed systems. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 291–309, Santa Clara, CA, 2016. USENIX Association.

[98] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, Hrishikesh B. Acharya, Kyriakos Zarifis, and Scott Shenker. Troubleshooting blackbox SDN control software with minimal causal sequences. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 395–406, 2014.

[99] Selendroid. Selendroid - http://selendroid.io/.

[100] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 391–406, 2013.

[101] Pivotal Software. Spring framework - https://spring.io.

[102] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4F: Taint analysis of framework-based web applications. In *OOPSLA*, pages 1053–1068, 2011.

[103] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 28. MIT press, 1998.

[104] Sriraman Tallam and Rajiv Gupta. Unified control flow and data dependence traces. *ACM Transactions on Architecture and Code Optimization*, 4(3), 2007.

[105] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *PLDI*, pages 87–97, 2009.

[106] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989.

[107] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, pages 13–23, 1999.

[108] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *TACAS/ETAPS*, pages 461–476, 2005.

[109] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 131–144, 2004.

[110] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83, 1945.

[111] Xamarin. Calabash - http://calaba.sh/.

[112] Tao Xie, Evan Martin, and Hai Yuan. Automatic extraction of abstract-object-state machines from unit-test executions. In *ICSE*, pages 835–838, 2006.

[113] Rubin Xu, Hassen Saïdi, and Ross J. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 539–552, 2012.

[114] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, pages 282–291, 2006.

[115] Wei Yang, Mukul R. Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 250–265, 2013.

[116] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. Droidfuzzer: Fuzzing the Android apps with intent-filter tag. In *The 11th International Conference on Advances in Mobile Computing & Multimedia, MoMM '13, Vienna, Austria, December 2-4, 2013*, page 68, 2013.

[117] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999*, pages 253–267, 1999.

[118] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18-22, 2002*, pages 1–10, 2002.

[119] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.