Elix: Path-Selective Taint Analysis for Extracting Mobile App Links

Yongjian Hu*
Two Sigma Investments

Oriana Riva
Microsoft Research

Suman Nath
Microsoft Research

Iulian Neamtiu
New Jersey Institute of Technology

ABSTRACT

App links, also known as mobile deep links, are URIs that point to specific pages in an app. App links are essential to many mobile experiences: Google and Bing use them to link search results directly to relevant pages in an app and apps use them for cross-app navigation. However, app links are hard to discover and, since they must be explicitly built into apps by developers, only exist for a small fraction of apps. To address these two problems, we propose *Elix*, an automated app link extractor. We define link extraction as a static information flow problem where a link, with its scheme and parameters, is synthesized by analyzing the data flow between subsequent pages in an app. As static analysis is prone to false positives, Elix adopts a novel, *path-selective* taint analysis that leverages symbolic execution to reason about path constraints and abandon infeasible paths. Elix can automatically and correctly discover links that are exposed by an app, and many others that are not explicitly exposed, thus increasing coverage of both link-enabled apps and link-enabled pages in an app. Elix also simplifies the scheme of extracted links by reducing complex types to a minimal set of primitive types. We have implemented Elix on Android and applied it to 1007 popular Android apps. Elix can extract 80–90% of an app's links, and above 80% of the extracted links are stable.

CCS CONCEPTS

• **Human-centered computing** → *Ubiquitous and mobile computing systems and tools*; • **Software and its engineering** → **Automated static analysis**.

KEYWORDS

Mobile app links; static analysis; symbolic execution

ACM Reference Format:
Yongjian Hu, Oriana Riva, Suman Nath, and Iulian Neamtiu. 2019. Elix: Path-Selective Taint Analysis for Extracting Mobile App Links. In *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19), June 17–21, 2019, Seoul, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307334.3326102

*Work done while at Microsoft Research.

MobiSys '19, June 17–21, 2019, Seoul, Republic of Korea

ACM ISBN 978-1-4503-6661-8/19/06…$15.00
https://doi.org/10.1145/3307334.3326102

1 INTRODUCTION

App links [2, 6, 17, 20], also known as mobile deep links, are URIs for specific pages within a mobile app. For instance, the Spotify app exposes the link "spotify:album:<album-id>"; following that link takes directly to the page for the album "album-id" in the app. Just as web links enable many indispensable user experiences on the web, app links promise the same for mobile apps. Google and Bing use app links to link search results to relevant pages inside apps; intelligent assistants use them to delegate user queries to specific app functionality (e.g., Google Assistant answers the query "Which movies are playing?" by retrieving movie showtimes via the IMDB app link imdb:///showtimes?date=2018-12-23); and apps and webpages use them to deep link to specific pages in an app (e.g., Google Maps directly invoking Uber's ride request page).

App links are also appealing to app developers, for various reasons: improved app navigation, improved user retention, and increased app discoverability [24]. These benefits are so crucial that app developers even pay for various deep linking services (Branch, Firebase, etc.), and they pay other apps to integrate their app links (e.g., Uber paid $5 for each new rider brought via app link [49]). However, despite their promise, the full potential of app links is yet to be realized. While app stores contain millions of apps, only a small fraction are linked to by other apps and services [38].

A main reason behind such low coverage is that *discovering* "usable" app links at scale is hard. Although app links must be declared in the app manifest file, the reported URI schemes often omit parameters that are required to invoke the links, rendering them useless. As a consequence, current practice for discovering an app's links is to analyze its corresponding website [27, 36, 37]. This is mainly because developers who want to make their app contents linkable in search engine results are encouraged to have a website mirroring the app content and reference the URI schemes specified in the app in each corresponding webpage [3]. Since many apps do not have a website counterpart (or they do, but developers do not annotate them with app links), this approach of discovering app links from webpages yields low coverage (§2.2). Moreover, app links discovered from webpages are often inaccurate, because developers do not keep them current as the app evolves.

Another reason for low coverage is the *lack of app links*. Unlike web deep links, exposing app links requires some developer effort (45–411 lines of code for a single link, according to [29]). Moreover, inadvertently exposing a link can be a security risk [28] or forgetting to update a link upon an app update makes it undiscoverable and can render the app/website unstable [32]. Consequently, few Android/iOS apps expose app links, and when they do, they only expose a couple of pages within the app; non-popular apps tend to not expose app links at all [8, 31]. As our study has revealed, even among the top-1000 Android apps, 55% expose no app links, 20% expose just one link, and only 25% expose more than 2 links (§5.1).

To address both problems – discovering existing app links and the scarcity of available app links – we propose the *Elix* app link extractor. For discovery, Elix statically analyzes an app binary to automatically extract its app links, a process that can scale to a large number of apps, including apps without webpages or that do not list app links on their webpages. To address the scarcity challenge, in addition to the links exposed by the app developer, Elix provides developers with an automated approach to build *new* links and publish them, if the are willing to do so. Thus, Elix increases *app link coverage* of both link-enabled apps (i.e., apps whose developers have manually exposed some app links) and of link-enabled pages inside each app (i.e., pages that the developer has not explicitly exposed).

Elix enables this double functionality with three goals in mind: *(i) Coverage*: extracting app links from any app and for all pages in an app, regardless of which app links have been manually exposed by the app developer. *(ii) Path validity*: extracted app links must trigger feasible execution paths in the app. *(iii) Practicality*: Elix-extracted app links should be amenable to be used as currently-known app links. While static app analysis can efficiently achieve the first goal, achieving also the other goals is challenging.

First, static analysis is prone to false positives, i.e., it can produce app links for execution paths and behaviors that are not actually feasible at runtime. To address this issue, Elix uses a novel mechanism, *path-selective taint analysis*, that leverages symbolic execution to reason about path constraints and abandon infeasible paths. Our path selectivity combines a precise static data-flow (taint) analysis with the control-abstraction power of symbolic execution, which yields an effective and efficient link extraction approach. Additionally, every extracted app link is tested using parameters automatically discovered through dynamic analysis.

Second, in a mobile app, a page (*Activity* in Android) is designed to be invoked by another page by providing some parameters. For instance, a page showing a restaurant's information may be reached from the search page by passing a restaurant identifier or a Restaurant-type object or some other internal variables. To extract an app link such as the one for the restaurant information page, Elix essentially needs to convert an "internal" interface between two app pages into an external interface invokable through a parametrized link. An internal interface is likely to take as input one or more objects with a complex type (tens of fields, potentially) requiring custom serialization. Due to the complexity and number of such input objects, directly including them as link parameters may make the link hard to use, falling short of our practicality goal. Elix therefore simplifies extracted links by automatically reducing parameters of complex objects to the *minimal set of required primitive types*.

In summary, we make the following contributions. *(i)* A static analysis-based approach and tool to automatically discover exposed and non-exposed Android app links. We improve on the state of the art for program analysis with *selective path analysis*: combining traditional data flow analysis with symbolic execution to reason about constraints on the structure of the input taint, hence increasing precision without sacrificing efficiency (§3). *(ii)* A practical framework to extract, test and validate app links, that can be deployed in existing app stores (§4). The current prototype has been implemented for Android. *(iii)* An evaluation on 1007 real-world Android apps where Elix provides 80% coverage or higher to 87% of apps; an in-depth analysis of 100 apps (§5); a comparison with web crawlers-based techniques for app link discovery, where Elix achieves 6x higher coverage (§2.2).

2 BACKGROUND, MOTIVATION AND GOALS

We motivate the need for app link extraction at scale with a few concrete use cases. We then discuss limitations of existing solutions and our goals. We use Android as a reference platform, but our observations extend to other platforms.

2.1 Target consumers of app links

We have designed Elix with three classes of consumers in mind: search engines, app developers, and app stores.

Search engines. App links enable indexing and searching of app contents, so that clicking on a search result on a mobile device takes the user directly to the relevant page in the app. Ideally, search engines would like to index all contents in an app, and hence they would like to have app links to most pages in an app. Search services that allow users to search app content they visited in the past (e.g., "Stuff I've Seen" [15]) also require app links. To discover app links, Google [36], Bing [37] and App Search APIs like URX [27] rely on web crawlers, under the assumption that developers have added app URIs as metadata in the app's corresponding webpage, which is a requirement to support app indexing [3, 27, 36].[1] This approach cannot discover app links exposed by mobile apps that do not have a web counterpart, and, as we will show shortly, even when apps have a corresponding website, it can discover only a tiny portion of the developer-exposed app links.

In-house app developers. Compared to web URLs, app links require work to be exposed and are more fragile. Exposing an app link entails: registering the link URI in the app manifest, then adding supporting code in the app to validate inputs and launch the requested page. Others have found that an average of 45–411 lines of code had to be changed to enable a new app link [29]. More importantly, developers must test and possibly modify the exposed links every time the app is updated. Elix can bring two advantages. *(i)* Each time the app is updated, developers can use Elix to verify which links are exposed, whether the URI formats declared in the app manifest are correct, and whether the links execute correctly. *(ii)* Elix discovers links to *all* pages in the app, not just those that the developer handled and declared in the manifest; using Elix developers can automatically obtain schemes and parameters for all links the app could expose, thus reducing the effort when exposing new links. In a nutshell, Elix helps developers the way compiler warning/suggestions do – as a warning "you are exposing this link" or as a suggestion "if you want to expose this link, do this".

App stores. Currently, app stores cannot extract, leverage, or validate app links: there is no repository of app links[2], and with the exception of few apps that publish up-to-date app link information for developers, link discovery is flawed. As we read in developer

[1] Of course, developers of apps without a website can index the app by themselves, communicate the index to search engines and update it periodically, but the effort is high. For doing this, Google provides Personal Content Indexing [19]. Out of the 1007 tested apps, 8 supported it.

[2] To the best of our knowledge, www.gotschemes.com is the only available repository, but it contains links for few apps without any guarantees on them being up-to-date.

Table 1: Comparison of web-based discovery and Elix.

App	Exposed links	Web discovery	Elix discovery
Open Table	10	1	10
Kayak	10	3	10
IMDB	23	0	23
Fandango	15	0	15
Shazam	15	3	15
Eat24	2	2	2
Dictionary.com	8	0	8
CNN	6	2	6
Duolingo	10	0	8
Zomato	10	1	10
Airbnb	19	8	19
ABC news	3	1	3
BBC news	1	0	1
AirWatchESPN	3	0	3
Average (%)	**9.6 (100%)**	**1.5 (16%)**	**9.5 (99%)**

Table 2: App link declarations in Android apps. In each example, the first row reports the link's target activity and parameters, and each of the following rows reports the URI format(s) declared in the app manifest. Parameters are often missing from the URI (e.g., date **and** movieId **in the 2nd link) and multiple URIs (2nd and 3rd links) are often specified for the same activity, but they have identical functionality (i.e., they point to the same page in the app).**

Zillow: HomeDetailsActivity:[itemId] *% get house information*
<data host=www.zillow.com pattern=/homedetails/.* scheme=http/>
IMDB: ShowtimesActivity:[date, movieId] *% get movie showtimes*
<data host=" prefix=/showtimes scheme=imdb/>
<data host=*.imdb.com prefix=/showtimes scheme=http/>
OpenTable: RestaurantProfileActivity:[restaurantId] *% get rest profile*
<data host=m.opentable.com prefix=/restaurant/profile scheme=https/>
<data host=m.opentable.co.uk prefix=/restaurant/profile scheme=https/>
<data host=www.opentable.com prefix=/restaurant/profile scheme=http/>
... (28 definitions for the same link)

forums, *i)* it is hard to find out about links exposed by other apps, even top ones [40, 44, 45, 48]; *ii)* even when discovered, it is hard to understand how to parameterize the links [39, 41, 42, 47]; and *iii)* there is a need to facilitate developer requests for new app links [43, 46]. Elix could be integrated in app stores as follows: when a developer submits a new app or updates an existing one, Elix analyzes the app and solicits developer consent to publish some or all of the automatically-extracted links. By integrating Elix, app stores can enable discovery and testing of app links at scale and incentivize developers to expose more app links. At the same time, developers retain control on which links are published.

2.2 Problems addressed by Elix

Elix addresses two fundamental problems in the mobile app ecosystem: *(i)* exposed app links are hard to discover, and *(ii)* few app links are generally exposed by app developers.

P1: Discovery. Current practice to discover app links relies on developers publishing app links on the app's webpage or in a sitemap file [3, 6, 18, 27]. If they do so, the app is added to Google's index and will appear in search results. Consider, for instance, the OpenTable app and its website. The http://www.opentable.com/capo webpage's source contains "android-app://com.opentable/vnd.opentable.deeplink/opentable.com/restaurant/profile?refId=12415&rid=202" which is in fact the app link to open the OpenTable app on the Capo restaurant page (the tag android-app is used to identify Android app links). Yet, that is the *only* link listed on the entire website; many links like /ratings/showratings, /search/results or /reservation that are exposed by the app are missing from the website. On the other hand, sitemap files, if used, usually list more app links but they omit link parameters, thus making the links unusable. Generally, publishing app links on webpages is not popular. More than two-thirds of top ranking websites with an app do not use app indexing [38], and, by definition, this approach does *not* work for apps without a website.

We verified our claims with an in-depth analysis of 14 popular Android apps which list various app links in their manifest file. As shown in Table 1, we discovered app links for these apps using three approaches: (i) to obtain a ground truth on the links exposed, we examined the app bytecode to extract the full URI format of all links listed in the manifest and to identify which UI page they reference (column 2 in the table); (ii) we manually searched for app links listed in the page sources of the websites of these apps (column 3); and (iii) we used Elix (column 4). Web discovery was able to find only 16% of all exposed app links. Instead, Elix was able to correctly discover links for all exposed pages, except for 2 in the Duolingo app (due to static analysis timeouts).[3]

P2: Coverage. Elix is useful not only for discovering currently-exposed links, but also for discovering new ones. Still a minority of Android and iOS apps expose app links. In Azim et al. [8], 23% of 14k top Android apps were found to expose some app links (typically fewer than 3), whereas non-popular apps do not expose any links. We counted the number of app link URIs in the manifests of 1007 most popular Android apps on Google Play in July 2017. This number represents an overestimate of the number of pages reachable in the app because apps often list multiple URI formats for links pointing to the same page (e.g., in Table 2 imdb:///showtimes and http://*.imdb.com/showtimes are different URIs, but point to the same page). We found that 55% of apps list no URIs, 20% declare one, and only 25% declare more than two. The average number of unique pages in these apps is 38 which means that even for apps exposing app links the majority of the pages in the app are not linkable.

2.3 Goals and possible solutions

To address the current situation, we propose automated *app link extraction*: we seek to extract *valid* app links that guarantee *good coverage* of app pages, and are *practical* to use. We discuss possible approaches to achieve this goal. The discussion is summarized in Table 3.

[3] Elix's discovered links have URIs slightly different from those listed on the app websites, but they serve the same purpose. More examples of Elix-extracted app links will follow later in this section.

Table 3: Possible approaches for discovering developer-coded app links and extracting new app links.

Approach	Discovery of links		Properties		
	exposed	new	Coverage	Validity	Practicality
Manifest-based discovery	Yes	No	Low	Yes	No
Dynamic analysis	Yes	Yes	Low	Yes	No
State-of-the-art static analysis	Yes	Yes	High	No	No
Elix	**Yes**	**Yes**	**High**	**Yes**	**Yes**

Manifest-based discovery. Exposed app links must be declared in the app manifest, so one scalable way to discover developer-coded app links is to statically analyze manifest files. However, manifest files typically contain "URI patterns" rather than complete definitions of app links. Such patterns often omit parameters that are required to invoke the links, rendering them useless. Table 2 shows some examples. The manifest of the IMDB app, for example, lists the /showtimes link for retrieving a movie showtimes for a certain date, but omits the required date and movieId parameters. Likewise, the OpenTable's restaurant/profile app link omits the restaurantId parameter.

Dynamic analysis. App link parameters could be intercepted at runtime, at every page transition, but this would require automatically running every app and visiting all its pages. This approach is unlikely to scale because dynamic app analysis suffers from poor coverage – measurements studies [12] show that with existing tools only 40% of an app's pages can be automatically visited. Moreover, abstracting from raw app logs to a usable and concise URI format including parameter types and constraints is not always feasible, unless an extensive collection of traces is available. As an example, Figure 1 shows the message intercepted when loading MapActivity in the Android OpenTable app (this message is called intent in Android). In intent.bundle one can identify the parameters and their values, such as restaurant and streetViewExtra used to create the target page. However, a large part of the intercepted parameters are not essential; in this case, only two are actually needed to invoke the page, as we show later.

```
intent.bundle:{
  restaurant:{type:com.opentable.models.Restaurant(@Parcelable),
              value:{"address":"3649 Mission Inn Ave",
                     "city":"Riverside",
                     "phone":"8883264448",
                     "profilePhoto": {"id":"23674228",...},
                     "restaurantId":150262,
                     "restaurantName":"Bella Trattoria Restaurant",
                     ...}
             },
  streetViewExtra:{type:java.lang.Boolean,value:true},
intent.action: null,
intent.uri: null,
intent.component: ComponentInfo{com.opentable/com.opentable.
                    activities.restaurant.info.MapActivity}
```

Figure 1: Example of intent intercepted at runtime in the OpenTable Android app. The highlighted parameters are the only ones that the target page actually requires to be launched. Elix correctly identifies them.

State of the art static analysis. Another approach is to rely on static analysis. Unlike dynamic analysis, static analysis provides good coverage, so an alternative for capturing page parameters would be to trace such parameters statically. However, static analysis suffers from false positives (i.e., some of the paths it identifies might be infeasible hence could fail to provide reliable app links) and precision vs. scalability trade-offs (i.e., we can have an efficient but imprecise analysis, or a precise but inefficient one). Moreover, because we are effectively trying to extract an internal app interface that the developer did not specify to be invoked externally, even though static analysis could capture it correctly, its API might not be practical. An internal interface is likely to take as input large objects with tens of fields and require custom serialization. Instead, we want to produce app links with straightforward schemes.

Elix's static analysis. To address these problems, simply capturing page parameters is insufficient. Instead Elix discovers how these parameters "flow" in the code generating the page (e.g., some parameters could be left unused, or others might be mutually exclusive). Going back to the previous example of OpenTable's MapActivity, Elix extracts the first link shown in Figure 2. The restaurant parameter in the intent in Figure 1, which dynamic analysis would capture, is replaced by string:restaurantName, which is the only field (out of the 61 com.opentable.models.Restaurant contains) really needed to invoke the link. Hence, Elix significantly simplifies the link scheme compared to what dynamic analysis or standard static analysis would output. Another advantage of Elix is its ability to identify parameter constraints thus ensuring extracted links execute reliably. Figure 2 shows more examples of Elix-extracted app links, and §5.4 gives examples of links with constraints.

3 EXTRACTING APP LINKS USING STATIC ANALYSIS

We describe Elix's design. To address the aforementioned challenges, Elix makes two key advances: *i)* it introduces a novel combination of taint tracking and symbolic execution which enables precise yet scalable analysis; and *ii)* it tackles the intricacies of the Android platform to permit analysis of real-world, substantial Android apps. We have chosen Android for its popularity; other phone platforms follow a similar model and likely exhibit similar challenges.

We first present an overview of the Android system and apps. An Android app consists of a collection of *activities* (pages), where each activity represents a single screen with a UI. Each activity is independent of the others. While activities are usually invoked from within the app, developers can set some activities as invokable from a different app, by exposing an app link, called *deep link* or dynamic link in Android. Activities are activated by an asynchronous message called an *intent*. For example, an app can send an intent to the Camera app, asking it to take a picture; or an app can post

```
# OpenTable - view restaurant information on a map
Activity: com.opentable.activities.restaurant.info.MapActivity
p1: string:(com.opentable.models.Restaurant)_restaurant{
    restaurantName}  // complex object reduction
p2: bool:EXTRA_ENABLE_DINING_MODE_UI
p3: bool:streetViewExtra
# OpenTable - submit restaurant review
Activity: com.opentable.activities.review.SubmitReviewActivity
p1: string:EXTRA_RESTAURANT_NAME
p2: string:EXTRA_EMAIL
p3: int:(com.opentable.dataservices.mobilerest.model.Review)_
    EXTRA_REVIEW{restaurantid}  // complex object reduction
# IMDB - get movie information
Activity: com.imdb.mobile.activity.FragmentTitleActivity
p1: string:com.imdb.mobile.tconst
# IMDB - view showtimes for a movie on a day
Activity: com.imdb.mobile.showtimes.ShowtimesActivity
p1: string:com.imdb.mobile.tconst
p2: string:com.imdb.mobile.date
# IMDB - get the profile of a celebrity
Activity: com.imdb.mobile.activity.FragmentNameActivity
p1: string:com.imdb.mobile.nconst
# BBCNews - get a collection of news
Activity: bbc.mobile.news.v3.app.CollectionActivity
p1: string:title
p2: bool:from_push
p3: string:uri
# BBCNews - get a piece of news
Activity: bbc.mobile.news.v3.app.ItemActivity
p1: string:uri
p2: int:pager_index
p3: string:title
```

Figure 2: Examples of app links extracted by Elix. For each link, we report the activity name and the list of parameters (type:parameter_name). The first two examples show links where Elix was able to reduce complex objects to the required primitive types (e.g., restaurantName of type string is extracted from restaurant of type com.opentable.models.Restaurant).

an item on Facebook by sending an intent to the Facebook app. An intent defines the *action* to perform and may specify the URI of the *data* to act on. Activities are listed in the app's manifest file. For each activity, the developer can specify one or more *intent filters* that declare the capabilities of the activity so that it can respond to intents from other apps. To expose a deep link, the activity's intent filter must include one or more <data> tags, where each tag represents a URI format, as shown in Table 2.

3.1 Overview and challenges

Taint analysis is used to track the propagation of information from a *source* to a *sink*. It is widely used in security, e.g., to find out whether information from a privacy-sensitive source, such as the user's GPS location, is sent, potentially after some processing, to an insecure sink, such as an advertising server [13]. Taint analysis can be dynamic or static. Dynamic taint analysis usually requires an instrumented platform to add "taint tags" that track data [16, 53]; dynamic approaches, however, have coverage problems as they rely on high-quality inputs for good coverage (and require instrumentation which may perturb normal app execution). Static taint tracking [7], on the other hand, is sound and scales well, but is prone to false positives and presents a precision–scalability tradeoff.

Example: Link extraction for a restaurant app. We illustrate the challenges in using static taint tracking for app link extraction using a sample restaurant app. Figure 3 (left) shows the source code. An activity for showing restaurant information receives a request to start (sent from the previous activity the user was on). The request's parameters are passed via an Intent object (line 2). The activity receives the information that will be displayed on the screen from a Bundle-type object contained in intent (a Bundle is a key-value store, essentially). If this is not the first time the activity is started, bundle is retrieved from the persistent storage savedInstanceState, which basically means the activity will show the same restaurant as in the previous run. On line 9 the activity starts looking into the request's parameter intent.getAction(). Lines 10–16 handle the case of a request with action=VIEW. The app fetches the restaurant complex object from the link parameters and displays the restaurant name (line 12), and then sets the Google Maps location to the restaurant's location loc (line 15). Lines 17–22 handle the case of a request with action=SEARCH: a list of nearby restaurants is extracted from the bundle and iterated upon (the "for" loop on lines 20–22).

Link extraction desiderata. Each time an activity must be started, an intent carrying the parameters necessary to its construction is passed to it. *The carried parameters and how they are used in the activity code implicitly define one or multiple paths to start that activity. Elix identifies and saves these paths in the format of app links.* Based on the code above, an app link extractor should: *i)* accurately extract *separate* app links for the View and Search functions, and *ii)* for each link, simplify the inputs by reducing the number of object fields (e.g., the restaurant object) so the app link can be invoked more easily (but still reliably).

Why is taint analysis alone inadequate? The information flow inferred by a state-of-the-art, but not path-selective, static taint analysis [7] is shown in Figure 3 (center). We indicate the data flow with arrows and the tainted data with an '*'; sinks and sources are marked explicitly. Numbers on the right indicate the position in the source code. Taint analysis is configured so that intent (on line 2) is marked as a SOURCE, and everything coming out of bundle as a SINK. The analysis of which parameters flow and how they flow from SOURCE to SINK will inform the app link generation. Since intent is tainted, line 7 will taint the bundle thus this taint will propagate to the intent.getAction() branches on lines 9 and 17. Moreover, all objects extracted from the bundle, which were marked as sinks, will be tainted: rid (line 10), restaurant (line 11), loc (line 14), query (line 18), and nearby (line 19). The problem occurs on line 23 where lack of path selectivity between the line 9 and line 17 branches *conflates* all 6 fields into a single 6-field object (action, rid, restaurant, loc, query, and nearby), instead of the correct 2-object extraction with 4 fields (action, rid, restaurant, loc) and 3 fields (action, query, nearby), respectively. Therefore, even an advanced static analysis would *incorrectly* extract a *single* app link with six parameters. This is a false positive. Attempting to invoke such a link may produce an incorrect result or fail.

How Elix correctly extracts the app links. In §3.2.4 we explain Elix's process (Figure 3, right), and how it correctly extracts two distinct app links. To do so and to meet the desiderata above, Elix introduces two new techniques: *i)* Path-selective taint tracking with symbolic execution, and *ii)* Complex object reduction.

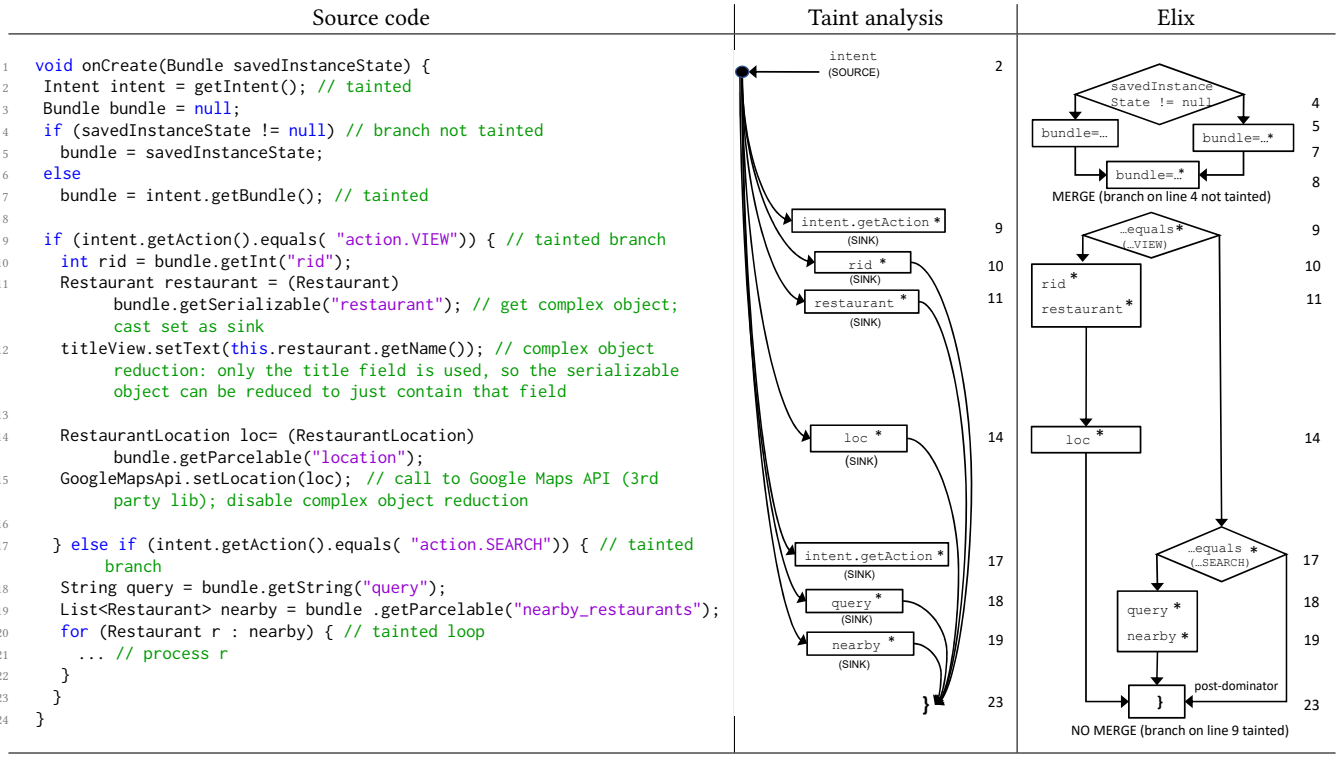


Figure 3: Source code (left); static analysis-inferred information flow (center); Elix path-selective information flow (right).

3.2 Elix design

Figure 4 shows Elix's architecture. In the following we describe all its components.

3.2.1 Setting up the static analysis.

Harness generation. Static analyses need starting (entry) points identifying where the execution starts. Android apps, unlike traditional Java programs, do not have a main() method that would serve as single entry. Rather, each activity is managed by the Android framework by invoking the activity's lifecycle callbacks, e.g., onCreate(), onStart(), etc. Elix synthesizes *harnesses* ("mock" entry points in a mock main() method) for this purpose.

Note that, as shown in Figure 4, Elix analyzes each app activity, activity_1 . . . activity_n, separately. This is possible because activities are self-contained, and advantageous as it keeps the approach scalable (approaches that perform whole-app analysis have scalability issues due to large memory and time demands [26, 50, 52]).

Taint propagation. Interest in Android security has led to many static taint trackers for Android, e.g., FlowDroid [7], Amandroid [50], or DroidSafe [26]. To leverage such trackers, our insight is that *app link discovery can be encoded as a taint tracking problem*, and *imprecision can be alleviated with path-selectivity via symbolic execution*. Since taint trackers look for information flow (taint) from a source to a sink, Elix defines a set of sources and a set of sinks. Intents are used to pass parameters between activities, and each activity is associated with an intent, so we define Activity.getIntent() as a *source*. For *sinks*, Elix finds which parameters have been fetched from the intent by activity code. The parameters inside an intent include action, extra bundle data, data uri, etc. We set all the data fetch API methods as sinks, such as intent.getData(), intent.getIntExtra(String key), etc. (more details in §3.3).

Taint propagation is based on the IFDS algorithm [34] whose complexity is polynomial. Elix runs taint propagation until converging (reaching a fix-point) and saves the result into a taint summary cache. Later, the symbolic executor queries the cache in constant time to get the taint status of each variable during path exploration. Taint summaries play a key role in the performance of path-selective analysis.

Taint summaries. Taint summaries save the taint status of variables for each analyzed method. The summary uses the format <sp,d1> → <n,d2,{d3}>, where sp is the start point of the method, and d1 is the tainted status for a method input parameter; n is the current statement, d2 is the incoming taint, and d3 a set of outgoing taints. The summary's semantics is: if at the entry point of a method, d1 holds, then at statement n, d2 also holds; after executing d2, it will produce a set of d3 values. This summary representation is inspired by the classical IFDS framework [34] and FlowDroid's data-flow representation [7]. For the code in Figure 3, an example of taint summary is <onCreate,0> → <line2,0,{intent*}>, where <onCreate,0> means at the entry of method onCreate the taint state is empty, and <line2,0,{intent*}> means at line 2 the in-taint state is empty but the out-taint state is {intent*} because getIntent() is the taint source. Another example is <onCreate,0> → <line10, {intent*, bundle*}, {intent*, bundle*, rid*}> meaning at line 10 the in-taint states are intent* from line 2 and bundle* from line 7; the

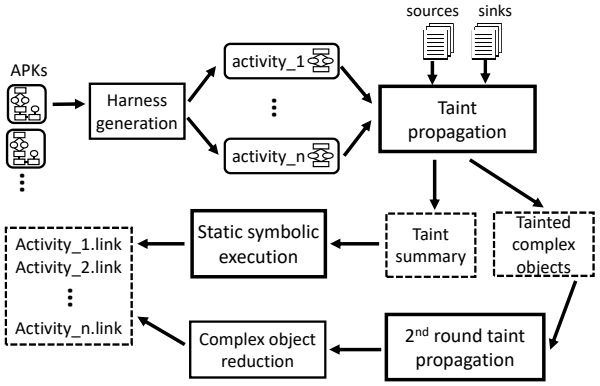


Figure 4: Elix architecture.

out-taint states are intent* and bundle* from the in-taint states and rid* generated at line 10.

Summaries play an important role in our analysis as they prevent redundant analysis of a method with the same taint parameters. An analysis tool can directly query the taint summaries of the onCreate function at line 10 and know how many variables are tainted and what the chain of taint propagation is. In the example of rid*, the propagation chain is intent* → bundle* → rid*. We leverage taint summaries for more precise path constraints analysis, as described shortly.

3.2.2 Path-selective taint tracking via symbolic execution.

Static taint analysis can be imprecise if path conditions (constraints) are not considered. Figure 3 has illustrated how ignoring path conditions leads to incorrect links, because the analysis conflates results from separate branches (line 23). Elix avoids this imprecision through symbolic execution to achieve *path-selective taint tracking*.

Elix's symbolic executor. Symbolic execution is a technique that allows symbolic reasoning about program state (*symbolic state*) and conditions to be met for executing a certain path (*path constraints*). Variables whose values are known are said to have *concrete values*; variables whose value is not known are handled symbolically, e.g., we assign symbolic value X to program variable x. A *symbolic executor* "executes" the program in the symbolic domain, e.g., after program statement x=x+1 the new symbolic value of X will be $X+1$. Path conditions are logical ANDs among branch conditions encountered so far. A branch "forks" the path into two mutually exclusive new paths. For example, if the current branch condition is π:X>1 and the next statement is if (y>42), the branch forks π into *(i)* π_1:X>1 && Y>42 on the then branch and *(ii)* π_2:X>1 && Y≤42 on the else branch. As other examples, in Figure 3, the path condition for reaching line 7 is savedInstanceState == null; similarly, the condition for reaching line 18 is intent.getAction() == "action.SEARCH". Of course, these are simplified examples; actual path conditions are much more complex.

Elix encodes taint status and path conditions in the taint summary by keeping track of program counter, current method, and call stack in the symbolic state. The executor updates the symbolic state (including current path condition) after each statement. Unlike traditional interpreters, Elix's executor keeps the taint summary from the previous taint propagation stage. The taint summary indicates *i)* whether the current method's arguments are tainted, and *ii)* the taint state of the next statement. If the current statement is a taint sink, Elix's executor saves the sink and all the pre-conditions so far as path constraints for this sink. Later, it uses this information to generate an app link.

Elix path selectivity. As path forking occurs whenever a branch is encountered, symbolic execution can suffer severely due to path explosion. Inspired by ESP [14], Elix avoids this problem via a new path selectivity approach that uses *path merging* and *path killing*. The key idea is to only fork paths at branches that have dependencies to the input intent, i.e., tainted branches, and merging paths at untainted branches. Note that during taint propagation, IFDS aggressively merges taint properties at every branch exit causing false positives. Therefore, path-selective analysis only needs to handle those tainted branches to de-couple merged taint status.

Path merging. Each branch point b has a corresponding post-dominator pd.[4] The pair <b,pd> specifies the scope of the branch, and within this scope, all the statements are control-dependent on branch b. Elix uses the post-dominator pd as merge point. When all the paths reach their merge point, the executor starts merging them. Note that traditional symbolic executors never perform path merges, or use a limited version of path merge. In our merge algorithm, we generally merge paths, with one exception: we do not merge if either of the branch condition operand is tainted. This selective merging algorithm collects structural constraints for the intent that avoid conflict conditions due to the intent itself. In our experiments, we found that many branches are not tainted, thus Elix's merging is effective at avoiding path explosion.

Path killing. A path is killed at a statement (or branch condition) which invalidates the current path, e.g., an assertion or a setIntent(Intent intent) method that overrides the tainted intent with a fresh one. This is because we aim to find paths that are affected by the intent from the environment. In addition, Elix kills a path with branch conditions such as getIntent() == null or getIntent().getExtras() == null, corresponding to app links with no parameters.

3.2.3 Complex object reduction.

It is common for an activity to fetch complex objects from the intent. For example, APIs Serializable getSerializableExtra(String key) and Parcelable getParcelableExtra(String key) fetch objects that implement Serializable or Parcelable interfaces, respectively. Objects implementing Serializable and Parcelable interfaces can be very complex and with several tens or hundreds of primitive fields (e.g., the Restaurant class in the OpenTable example in Figure 1 has 61 primitive fields while the Reservation class in the Airbnb app has 1162). Exposing such a complex type in an app link would make the link hard to use, hence would not meet our practicality goal. However, in practice, only a few fields of these complex objects are used frequently. Based on this insight, Elix performs a *second round of taint propagation* by starting from the Serializable getSerializableExtra(String key) and Parcelable getParcelableExtra(String key) as the source of taint, and keeps track of the fields used. For example,

[4]For any statement b, its post-dominator pd is the first statement that will be executed on any path from b to program end.

if Elix finds a restaurant object fetched from getSerializable and of type Restaurant, and observes restaurant.getName() being accessed, then it knows that the name field of the Restaurant object is used. The accessed fields are saved and reported as part of the link indicating the potential usage of the complex object in the analyzed activity. If there are no accessed fields, the object is removed from the link parameters. Note that Elix simplifies the parameters in a conservative way. If a tainted complex object flows into a system method that exceeds Elix's analysis scope (e.g., Fragments), Elix assumes all its fields are used and stops simplifying.

3.2.4 *Putting it all together.*

Elix successfully meets the desiderata laid out in §3.1; the analysis process is shown in Figure 3 (right). Diamonds represent branch statements, arrows indicate data flow, and tainted data is marked with '*'. Note that Elix *merges the two paths* at line 8 because the branch at line 4 is not tainted indicating this branch is not related with intent structure constraints. As a result bundle on line 8 will be tainted. The branch on line 9, however, is tainted so on line 23 Elix *does not perform a merge*. Rather Elix *path-selects*, i.e., splits the analysis for the then/else branches on lines 9 and 17. So Elix extracts two links from the two separate branches. Elix also performs complex object reduction as follows: due to the getSerializable API (line 11) we know that restaurant is a complex object. Because only the name field is used (line 12), Elix simplifies restaurant to a single field, "name". However, the complex object loc cannot be simplified since it is passed to the framework (line 15) which could perform arbitrary processing on it. Therefore, so far Elix has extracted a link VIEW with parameters int:rid, string:restaurant.name and RestaurantLocation:location. Next, Elix analyzes the other branch (lines 17–22) and extracts a link SEARCH with parameters string:query and [Restaurant]list:nearby_restaurants.

3.3 Implementation

We wrote Elix's symbolic executor from scratch. To build Elix's taint analysis we leveraged FlowDroid [7], but modified it in various ways.

We provided FlowDroid with our list of sources and sinks. As all link parameters are extracted via Activity.getIntent(), we set this as the only *source*. For *sinks*, we need to consider all object types which may be stored in an intent: action, extra bundle data, data uri, etc. We set all the data fetch API methods for these objects as sinks (e.g., intent.getAction(), intent.getData(), etc.); Bundle.get*() are set as sinks too – if a bundle is fetched from a tainted intent, then the data fetched from it is also tainted (like in Figure 3). Note that having only one source significantly reduces the number of taints, hence reducing FlowDroid's running time.

Elix extends FlowDroid in several aspects so it can analyze an extensive number of real-world apps and support our usage setting.

First, Elix generates separate harness methods (modeling the activity lifecycle callback and UI callbacks) for each activity, instead of a single harness for all activities, as FlowDroid does. This is because we aim to generate links for each activity. Moreover, setting the analysis scope to individual activities saves memory because taint results are not shared across activities. Also, Elix's harnesses use an object-sensitive model which is more precise than FlowDroid's.

Second, we had to overcome several engineering challenges to be able to run Elix on commercial apps. For example, commercial apps can be spread across multiple .dex files, so we added support for multi-dex analysis. Another challenge was intermediate language (IR) ambiguity: FlowDroid uses Jimple, a 3-address non-SSA format, as their IR. The non-SSA language may introduce ambiguity during backward taint propagation because local register names may be reused – we modified the IR to eliminate such ambiguities.

Third, Elix extends FlowDroid's call graph to account for message passing: we add call edges upon discovering calls to the message-passing API, e.g., Handler.sendMessage(), Handler.post(Runnable), etc. Thus, Elix has better coverage than FlowDroid.

3.4 Limitations

Elix has three main technical limitations, which can be addressed with more engineering effort.

First, it does not handle *Fragments*. In Android, an Activity can include one or multiple Fragments, which are a kind of "sub-activities", each associated with a UI portion (e.g., tabs in a screen). A Fragment has its own lifecycle and receives its own input events. Elix does not yet model the Fragment lifecycle so false negatives can arise if non-tainted parameters are consumed by Fragments. More importantly, the presence of Fragments can limit the effectiveness of complex object reduction (§3.2.3). As reported in §5.2, while we have not seen many such occurrences these may be more common in the future.

Second, it does not handle *Dependency injection (DI)*. DI aims to make a software class more reusable and testable by decoupling hard dependencies between objects, e.g., instead of a class instantiating a specific logging service, that service is injected by an external framework. DI has been adopted also by mobile apps. DI poses static analysis challenges. Static analysis relies on "hard coded" object dependencies for pointer analysis and call-graph construction. To precisely know which object is injected, Elix would need to analyze DI code, which is tedious given the many existing DI frameworks, e.g., Dagger [23], Roboguice [21], ButterKnife [51]. Injected objects may contain code that uses tainted variables thus affecting Elix's precision.

Finally, dependencies between activities not captured through intent passing can also be problematic. If an app uses other ways (e.g., file system, network) to pass data between activities and those data dependencies are not captured by the input intent, the app links that Elix discovers may be in partial state or fail. Addressing this problem requires dynamically identifying such dependencies and replaying the whole sequence of activities as done in uLink [8] and Aladdin [29].

4 ELIX IN ACTION

We describe additional modules provided by the Elix framework to execute and test extracted app links.

4.1 Executing app links

In order to execute Elix-extracted app links, developers must take two steps. First, they need to update their app's manifest file with the URI schemes extracted by Elix, for all the new pages they intend to expose. Second, they need to add to their app the *Link Executor*

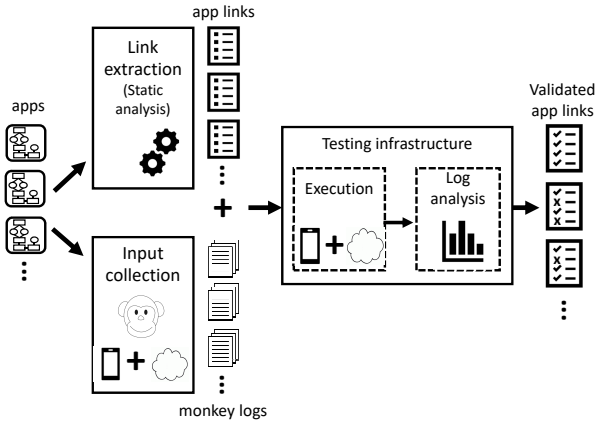


Figure 5: Testing infrastructure for validating app links.

module which takes care of assembling an Intent object from the specified link parameters and invoking the target activity, in the following way.

Given a parameterized app link, the Link Executor parses it, verifies that its constraints are satisfied, and generates an intent for its target activity. Links extracted by Elix use the URI format elix://<pkg>/<act-name>?[<params>]. The URI host points to the target app's package name (pkg) while the URI path contains the target activity name (act-name). Then the link contains query parameters (params), each formatted as <type>_<key>{<[field]>}=<value>, where the list of class fields ([field]) is present only if type is a complex object to which Elix applied input reduction. For example, the notation (com.opentable.models.Restaurant)_restaurant{restaurantName} means that the parameter with key=restaurant and of type=com.opentable.models.Restaurant has been reduced to the single field restaurantName. Possible link constraints are encoded separately as <type>_<key>=<condition>. Examples of Elix-extracted links are provided in Figure 2 and Figure 8.

For primitive-typed parameters (integers, strings, floats, etc.), the executor simply extracts the key and value, and saves them to the intent. For complex objects that implement the Parcelable or Serializable interfaces, the process is more involved. To simplify the link parameterization process, we allow an app or service invoking a link to specify *only* primitive types, thus not requiring it to obtain the APK of the target app. To do so, offline, Elix automatically executes every app (more details on this later) and inspects intercepted intents to extract the structures of complex objects. At runtime, the executor uses the class type specified in the invoked link to load the target app's APK dynamically and find the specific class structure for the object (i.e., object structures are logged in JSON format and are transformed into Java objects using Gson [25]). The fields corresponding to the keys passed in the link are then set and the whole object is saved into an intent. Finally, with the assembled intent, the executor invokes the target activity.

4.2 Testing app links

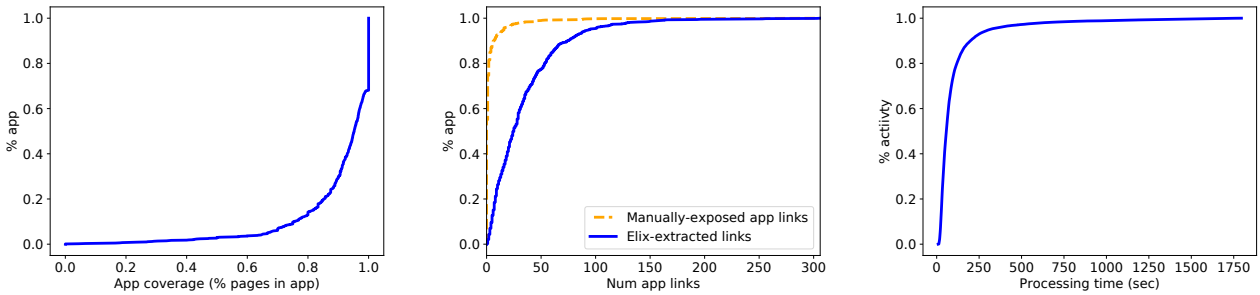
We built an automated pipeline (see Figure 5) that given a list of Elix-extracted links collects activity logs, parameterizes links, executes them, and classifies their result as success or failure. Apps are executed on physical phones (Nexus 5) and Virtual Machines (VMs). VMs are hosted in Azure Hyper-V and run Android X86 6.0 [5]. We modified Android 6.0 to log various information each time a new activity is loaded, as described later. Note that for our evaluation we tried to execute all Elix-extracted links. In reality, a developer would run through testing only the links that are already exposed or that have to be exposed.

Link parameterization. To discover app links, Elix does not require any dynamic analysis of the app. However, we use dynamic analysis for automatically logging structures of complex objects (as described above), testing extracted links and enriching the documentation of the link schemes with examples of usage. We use UI automation tools [12] to collect activity logs. These tools can be configured to simulate user interactions with an app (by tapping on buttons, filling out text boxes, swiping pages) and to navigate various pages. By default, we use the Android Monkey [4] and configure it to generate 35k random UI events (touch, scroll, swipe, etc.) with a delay of 500 msec between events. For apps requiring login, we create an account and save the credential state, which is restored at each Monkey run. Each time Monkey transitions to a new activity, we collect an example of app link invocation, including keys/values of its parameters (an example log is shown in Figure 1). Unfortunately, Monkey generally fails to visit all activities of an app, but parameter keys and complex objects are likely to recur in links for different activities of the same app, so even though Monkey is not able to visit all the activities, we can collect parameters for many of the extracted links. In this way, during testing, we try to parameterize any app link discovered by Elix

Execution and validation. Tests are launched from a PC connected to VMs/phones via ADB. We collect crash logs, intent signatures, UI trees including text contained in all textual UI elements, and page screenshots. We use this data to automatically classify each link as (i) a valid link that successfully executed, (ii) an invalid link that failed, or (iii) a link whose test outcome is unclear or that could not be parameterized automatically. Crash logs and presence/absence of intent dumps and a screenshot which are collected when a page loads are the most important features. In case of indecision we compare sizes of UI trees, amount of text and number of images against lower-bound thresholds. Screenshots are also useful for a quick manual inspection. For the evaluation in this paper, we used this testing infrastructure but we also randomly inspected many of the logs. In general, false negatives were rare. False positives occurred mainly when a link executed, but did not lead to the target page (e.g., a page with a popup error). Many improvements are possible here, such as a more advanced analysis of the collected screenshots.

5 EVALUATION

We evaluate Elix based on the goals we have set in §2.3: high coverage, link validity, and practicality. In addition, we show that our static analysis implementation is efficient and that testing of extracted links can be automated to scale to large numbers of apps. At the end of this section, we also provide a comparison of Elix with a "non-path-selective link extractor".



(a) App coverage **(b) Increase in number of links** **(c) Execution time per activity**

Figure 6: Static analysis coverage and execution time (based on 1007 top Android apps).

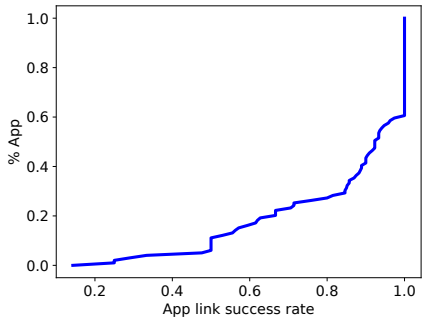


Figure 7: CDF of success rate for executed app links.

All experiments were executed using the testing infrastructure described in §4.2. Static analysis was performed on two machines: an Intel(R) Xeon(R) CPU E5-2687W v2 3.40GHz with 64 GB RAM and an Intel(R) Xeon(R) CPU E5-1650W v3 3.50GHz with 32 GB RAM. The differences between execution times on the two machines were negligible.

We used a total of 1007 Android apps, selected as follows. We identified the top 60 apps in 18 popular categories (excluding games) from Google Play Store and downloaded their APKs from https://www.apkmirror.com (which ensures that APKs are real and created by their respective developers). Some APKs were not available at this site, and hence some categories had fewer than 60 apps. The downloaded apps had an average size of the APK files of 20.9 kB, and had on average 38.2 activities.

5.1 Coverage and time of static analysis

We ran Elix with a maximum analysis time per activity of 30 minutes. Across the 1007 apps, out of 38,419 activities analyzed, 34,156 activities succeeded (88.90%), 3,900 timed out (10.15%), and 363 failed (0.95%). Timeouts mainly occurred in FlowDroid's taint propagation stage which is computation-intensive, and sometimes in Elix's symbolic executor. Failures were mostly due to corner cases which we do not handle yet, e.g., missing method body when applying complex object reduction.

Across all apps, Elix extracted a total of 57,750 links (on average 1.7 links per activity). Figure 6a reports the CDF of the percentage of app activities with app links extracted. 87% of apps have 80% coverage or higher (i.e., Elix extracted app links for 80% of an app's activities), and 32% achieve 100% coverage (i.e., Elix extracts links for all activities). We conclude Elix is able to extract app links for most pages in an app.

Moreover, as Figure 6b shows, Elix (blue line) significantly increases the coverage of an app's pages provided using today's developer-coded app links (orange dashed line). We estimated the number of developer-coded links by counting the deep link URIs declared in the app manifests. Keeping in mind this count is an over-estimate of the number of unique pages an app exposes (§2.2), we observe that 45% of the 1007 top Android apps expose 1 or more deep links and only 25% expose more than 2; Elix extracts more than 2 apps links for 98% of apps and more than 25 for 50% of apps.

App analysis times indicate Elix's efficiency. The average analysis time for successful activities was 103.9 seconds (maximum was 29.9 minutes); for 80% of the apps, analyzing a single activity took less than 2 minutes (Figure 6c).

5.2 Link validity

To verify that the paths extracted by Elix translate into practical links, we executed links from 100 apps. We used the testing infrastructure described in §4.2 to automatically collect instances of link parameters (using Monkey), generate parameterized links, and execute them in our testbed of phones and VMs. Across the 100 apps, Monkey provided an average coverage of 29.3% of an app's activities[5], which allowed us to automatically parameterize a total of 1,386 links (corresponding to 40% of all the links Elix extracted for the 100 test apps). These links were executed and, using the collected logs, classified as successful or failed. Links that could not be automatically parameterized were excluded from testing.

Overall, the average success rate per app was 84%. As we can see in Figure 7, for more than 80% of the apps the success rate was above 67% and for 50% of the apps was over 92%. We found three main causes of failure, listed in order of prevalence. 1) Dependency injection: the link was missing a required parameter due to

[5]This result is in agreement with an extensive comparison study [12] of popular UI automation tools.

Table 4: App link coverage (number of activities for which Elix extracted at least an app link compared to the total number of activities present in the app) and validity (how many of the tested links executed correctly) for 25 sample apps.

App	Category	Coverage		App link validity		
		# activities	# activities w/ link(s)	# tested links	success	failure
IMDB	Entertainment	67	67	14	12	2
Netflix	Entertainment	47	45	10	9	1
Spotify	Music&Audio	82	53	27	22	5
TuneIn Radio	Music&Audio	18	16	13	12	1
OpenTable	Food&Drinks	54	52	13	12	1
DominosPizza	Food&Drinks	37	37	24	23	1
Zomato	Food&Drinks	106	104	27	26	1
Wunderground	Weather	41	39	15	14	1
Kindle	Books&Ref	95	83	40	27	13
Dictionary.com	Books&Ref	29	28	19	18	1
CNN News	News	31	28	10	9	1
AllRecipes	Lifestyle	20	20	11	10	1
Walmart	Shopping	71	70	47	44	3
Wish	Shopping	45	37	34	32	2
Etsy	Shopping	36	36	26	22	4
Zillow	House&Home	47	47	23	20	3
Trulia	House&Home	29	27	10	10	0
Vine	Video	43	42	15	14	1
Instagram	Social	24	22	13	8	5
WebMD	Health&Fitness	38	38	20	17	3
Duolingo	Education	28	21	14	10	4
GasBuddy	Travel&Local	34	33	17	17	0
Hotel Tonight	Travel&Local	41	41	21	18	3
Tripit	Travel&Local	72	72	29	22	7
ESPN	Sports	39	26	15	14	1
Average		47.0	43.4	20.3	17.7	2.6

dependency injection not yet supported by Elix (see §3.4). 2) Link Executor limitations: links failed when i) they required parameters of type Bundle, ParcelableArray or ParcelableArrayList whose serialization/deserialization is not yet supported by our Link Executor, and ii) their parameters included field types of a class declared as a Java Interface so our Link Executor was not able to load the appropriate constructor (the constructor name should be captured during static analysis). 3) False negatives due to Fragments (§3.4). We conclude that Elix generated working links in the majority of cases, and that failures were due to tractable engineering issues.

For brevity we omit detailed results for all tested apps, but to give some concrete examples, Table 4 reports detailed results for 25 of the apps we executed. The table shows the number of activities for which one or more links were extracted (4th column) and the number of links tested (5th column) along with the outcome (last two columns). On average, 87% (17.7 out of 20.3) of the tested links worked reliably. We inspected the logs of the 65 failures across the 25 apps and found that dependency injection was the most common cause (31 out of 65), followed by Link Executor (18 out of 65) and Fragment limitations (12 out of 65). In four cases, the link failed due to app-specific issues (e.g., a system error or a dialog which could not be created).

Overall, Elix was able to successfully extract reliable links from many Android apps, from an extensive range of categories.

5.3 Reduction in link parameters

To demonstrate that Elix is effective at reducing the complexity of extracted links, we compared the link schemes before and after running the second taint propagation pass (§3.2.3), which attempts to minimize the number of complex-typed parameters. On average, complex data types were reduced by 44.9%. The removed/reduced complex objects were quite large. By examining the number of fields in the removed or reduced complex objects, we found the average reduction in terms of primitive types was 32.8 fields per each removed/reduced object, so a significant simplification in the link scheme. The number of complex types could be reduced even more by extending our taint propagation to Fragments.

5.4 Comparison with non-path-selective static analysis

To give some insights on the importance of Elix's novel path-selective approach, we implemented a non-path-selective link extractor that uses FlowDroid (with our modifications). We used four popular apps (Airbnb, Walmart, Netflix, and Fandango), and extracted links using both Elix and the non-path-selective extractor. In total, Elix generated 682 links for 297 activities (on average, 2.3 links per activity). The non-path-selective extractor generated 1

```
Elix
# C1 constraint
android.intent.action == "SEARCH"
# C2 constraint
(com.airbnb.utils.SettingDeepLink)_setting_deep_link == EXIST
# applink 1: search room listings, under C1=true
Activity: com.airbnb.activities.ManageListingActivity
p1: string:android.intent.action
# applink 2: update room details, under (C1=false & C2=true)
Activity: com.airbnb.activities.ManageListingActivity
p1: string:android.intent.action
p2: int:(com.airbnb.utils.SettingDeepLink)_setting_deep_link{ordinal}
# applink 3: view current listings, under (C1=false & C2=false)
Activity: com.airbnb.activities.ManageListingActivity
p1: string:android.intent.action
p2: Parcelable:(com.airbnb.models.Listing)_managed_listing
Non-path-selective extractor
# single applink without constrains which fails
Activity: com.airbnb.activities.ManageListingActivity
p1: string:android.intent.action
p2: Parcelable:(com.airbnb.models.Listing)_managed_listing
p3: Serializable:(com.airbnb.utils.SettingDeepLink)_setting_deep_link
```

Figure 8: Links and constraints extracted by Elix and a non-path-selective extractor for the ManageListingActivity of the Airbnb Android app.

link per activity, with no constraints. It incorrectly conflated all parameters of different paths in one path because it ignores control dependencies. This can lead to invalid links. For example, for Airbnb's ManageListingActivity, which is used for managing room listings by an owner, Elix extracted three links with 1 or 2 constraints each (shown in Figure 8). If android.intent.action equals SEARCH, the app link opens a page for searching through the owner's listings; otherwise, depending on whether the parameter setting_deep_link is specified or not, it opens a page for updating a room details or for viewing a list of rooms currently listed, respectively. The single link extracted by the non-path-selective extractor fails to capture these constraints and the three different functionalities. Moreover, Elix's links are simpler thanks to complex object reduction. We also found cases (e.g., Netflix's MovieDetailsActivity), where Elix was able to capture constraints about critical parameters that if not present or if set incorrectly would trigger exceptions.

The non-path-selective extractor ran faster than Elix. On average, the analysis of an activity took 37.8 seconds with the non-path-selective extractor and 49.1 seconds with Elix, but Elix's execution time was still acceptable.

6 RELATED WORK

The closest systems to Elix are uLink [8] and Aladdin [29]. Both uLink and Aladdin are frameworks to allow users to record links to specific states (i.e., activities or fragments) in an Android app. uLink records intents at runtime as opaque byte arrays, and it can only record links, effectively bookmarks, to pages a user has explicitly visited. Instead, Elix extracts parameterized URIs and achieves high coverage by using static analysis. uLink is also built as a library, thus requiring modifications to the app's source code. Aladdin exposes app links by synthesizing the shortest path from an app entry to the target activity with the help of a static activity navigation graph and a dynamically-constructed set of parameters to reach that activity. Aladdin's parameter construction is unsound because it only considers the shortest path to the target and the constructed parameters are only a subset of the total parameters. Compared to Aladdin, Elix's link parameter construction is sound since it relies on sound static analysis and the precision is improved by path-selective tuning. Aladdin also does not provide support for intents with Parcelable/Serializable objects. DroidLink [30] is somewhat similar to uLink and Aladdin: it allows activities to be accessed directly via app links by constructing an activity transition graph, computing the shortest path that allows a certain activity to be reached, generating a hash of the path and repackaging the app to accept these hashes via intent filters. DroidLink, just as uLink and Aladdin, is based on dynamic analysis, hence shares the same disadvantages. DroidLink has been tested on 5 apps only.

The idea of applying static taint analysis techniques to app link extraction is relatively new. Elix leverages FlowDroid; however, FlowDroid is path-insensitive, dataflow-oriented and produces false positives; we built a new static symbolic executor to remedy this. Traditional dynamic symbolic execution (DSE) [10, 11] leverages a mix of symbolic and concrete execution. DSE has several applications: higher code coverage, testing [10], input generation [22], security, etc. However, DSE can be limited by path explosion. Although progress has been made [33, 35], using DSE in large systems such as Android remains a challenge. For example, Anand et al. [1] used DSE to generate input events for app GUI testing; classical DSE did not complete for $k = 4$ event sequence within 12 hours, and even with their proposed pruning algorithms the length of the event sequence was bounded to $k = 4$. The main reason behind path explosion is the number of symbolic values selected, and the branches or loop conditions that are affected by the symbolic values because DSE forks a new path at each symbolic branch. Elix proposes static path-selective taint analysis that only forks paths at tainted branches and merges paths at post-dominators of untainted branches. Our approach is inspired by Thresher [9] and Saturn [52]. Thresher's goal is different from ours though, and it uses backward symbolic execution, while we perform forward symbolic analysis. Saturn uses method summaries for inter-procedural analysis and is path sensitive, but it is too heavyweight for complex systems. Instead, Elix takes a hybrid approach: it collects selective path constraints from plain dataflow-based taint results.

7 CONCLUSIONS

We have presented Elix, an approach that automates app link extraction. The approach hinges on a precise yet scalable path-selective static analysis that was achieved by combining taint analysis with symbolic execution. Our long-term goal is to allow developers to automatically create links for all app pages they want to expose. While there is still a long way to achieve this vision, our experiments on 1007 popular Android apps show that Elix works well for a variety of apps.

ACKNOWLEDGMENTS

We thank our shepherd, Lin Zhong, and the anonymous reviewers for their feedback. We also thank Jason Kace for his help in modifying the Android framework and building the cloud backend. The last author is partially supported by the NSF grant CNS-1617584.

REFERENCES

[1] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 59, 11 pages. https://doi.org/10.1145/2393596.2393666

[2] Android Developers. 2018. Enabling Deep Links for App Content. http://developer.android.com/training/app-indexing/deep-linking.html.

[3] Android Developers. 2018. Making Your App Content Searchable by Google. https://developer.android.com/training/app-indexing/index.html.

[4] Android Studio. 2018. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html.

[5] Android-x86. 2018. Porting Android to X86. http://www.android-x86.org/download.

[6] Apple Developer. 2018. App Search Programming Guide - Support Universal Links. https://developer.apple.com/library/content/documentation/General/Conceptual/AppSearch/UniversalLinks.html.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 259–269.

[8] Tanzirul Azim, Oriana Riva, and Suman Nath. 2016. uLink: Enabling User-Defined Deep Linking to App Content. In *Proc. of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '16)*. ACM, 305–318.

[9] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: Precise Refutations for Heap Reachability. In *Proc. of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. 275–286.

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, 209–224.

[11] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

[12] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *Proc. of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, 429–440.

[13] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. 2004. Understanding Data Lifetime via Whole System Simulation. In *Proc. of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, 22–22.

[14] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive Program Verification in Polynomial Time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/512529.512538

[15] Susan Dumais, Edward Cutrell, JJ Cadiz, Gavin Jancke, Raman Sarin, and Daniel C. Robbins. 2003. Stuff I've Seen: A System for Personal Information Retrieval and Re-use. In *Proc. of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval (SIGIR '03)*. ACM, 72–79.

[16] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*. USENIX Association, 393–407.

[17] Facebook. 2018. Applinks. https://developers.facebook.com/docs/applinks.

[18] Firebase . 2018. Firebase App Indexing. https://firebase.google.com/docs/app-indexing/.

[19] Firebase. 2018. Enable Personal Content Indexing. https://firebase.google.com/docs/app-indexing/android/personal-content.

[20] Firebase. 2018. Firebase Dynamic Links. https://firebase.google.com/docs/dynamic-links.

[21] Github. 2016. RoboGuice: a framework that brings the simplicity and ease of Dependency Injection to Android. https://github.com/roboguice/roboguice.

[22] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (Jan. 2012), 8 pages. https://doi.org/10.1145/2090147.2094081

[23] Google. 2017. Dagger: a fully static, compile-time dependency injection framework for both Java and Android. https://google.github.io/dagger/.

[24] Google. 2017. Driving installs and usage with Firebase App Indexing. http://firebase.google.com/docs/app-indexing/partners/case-study-one-pager.pdf.

[25] Google. 2017. GSON: A Java serialization/deserialization library that can convert Java Objects into JSON and back. https://github.com/google/gson.

[26] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *Proc. of the 22nd Annual Network and Distributed System Security Symposium, NDSS 2015*.

[27] InfoQ. 2018. Implementing and Searching Deep Links with the URX API. https://www.infoq.com/articles/urx-deep-links.

[28] Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, and Gang Wang. 2017. Measuring the Insecurity of Mobile Deep Links of Android. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 953–969.

[29] Yun Ma, Ziniu Hu, Yunxin Liu, Tao Xie, and Xuanzhe Liu. 2018. Aladdin: Automating Release of Deep-Link APIs on Android. In *Proceedings of the 2018 World Wide Web Conference (WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 1469–1478. https://doi.org/10.1145/3178876.3186059

[30] Yun Ma, Xuanzhe Liu, Ruogu Du, Ziniu Hu, Yi Liu, Meihua Yu, and Gang Huang. 2016. DroidLink: Automated Generation of Deep Links for Android Apps. *CoRR* abs/1605.06928 (2016). arXiv:1605.06928 http://arxiv.org/abs/1605.06928

[31] Marketing Land. 2014. Study: Only 22 Percent Of Top 200 Apps Using Deep Links. http://marketingland.com/study-22-percent-top-200-apps-using-deep-links-90177.

[32] Medium. 2017. Mobile Deep Linking Part 2: Using a Hosted Deep Links Provider. http://medium.com/ageitgey/mobile-deep-linking-part-2-using-a-hosted-deep-links-provider-c61fa58d083e.

[33] David A. Ramos and Dawson R. Engler. 2016. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*. https://www.usenix.org/conference/atc16/technical-sessions/presentation/ramos

[34] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. 49–61. https://doi.org/10.1145/199448.199462

[35] Anthony Romano and Dawson R. Engler. 2013. Expression Reduction from Programs in a Symbolic Binary Executor. In *Model Checking Software - 20th International Symposium, SPIN 2013, Stony Brook, NY, USA, July 8-9, 2013. Proceedings*. 301–319. https://doi.org/10.1007/978-3-642-39176-7_19

[36] Search Engine Land. 2015. App Indexing & The New Frontier Of SEO: Google Search + Deep Linking. http://searchengineland.com/app-indexing-new-frontier-seo-google-search-deep-linking-226517.

[37] Search Engine Land. 2015. Bing begins building index of native app content through app indexing. http://searchengineland.com/bing-begins-building-index-of-native-app-content-through-app-indexing-221374.

[38] Search Engine Land. 2016. Searchmetrics study shows most apps are not utilizing Google App Indexing. http://searchengineland.com/searchmetrics-study-shows-most-apps-not-utilizing-google-app-indexing-api- 244432.

[39] Stack Overflow. 2014. Spotify Deep link URI not working on Android Platform. https://stackoverflow.com/questions/26030629/spotify-deep-link-uri-not-working-on-android-platform.

[40] Stack Overflow. 2015. Deep link into eBay iOS app with custom URL scheme? https://stackoverflow.com/questions/31222583/deep-link-into-ebay-ios-app-with-custom-url-scheme.

[41] Stack Overflow. 2015. Deep linking to Facebook Messenger. https://stackoverflow.com/questions/31777075/deep-linking-to-facebook-messenger.

[42] Stack Overflow. 2016. Aliexpress deep link ios. https://stackoverflow.com/questions/41290803/aliexpress-deep-link-ios.

[43] Stack Overflow. 2016. Deep link to Medical ID tab of Health app? https://stackoverflow.com/questions/35469344/deep-link-to-medical-id-tab-of-health-app.

[44] Stack Overflow. 2016. How to deep link to OpenTable. https://stackoverflow.com/questions/31464943/how-to-deep-link-to-opentable.

[45] Stack Overflow. 2016. WhatsApp deep link to a specific mobile number. https://stackoverflow.com/questions/35995775/whatsapp-deep-link-to-a-specific-mobile-number.

[46] Stack Overflow. 2016. WIP - Add rental_url to free bikes and stations? https://github.com/NABSA/gbfs/pull/25.

[47] Stack Overflow. 2017. Uber deep-linking not working from website. https://stackoverflow.com/questions/42950122/uber-deep-linking-not-working-from-website.

[48] Stack Overflow. 2018. Open Walmart Affiliate Deep Link to cart of items in Walmart App. https://stackoverflow.com/questions/49680562/open-walmart-affiliate-deep-link-to-cart-of-items-in-walmart-app.

[49] The Verge. 2015. Uber wants to integrate with all your apps. https://www.theverge.com/transportation/2015/12/2/9835604/uber-button-app-developer-API-rideshare.

[50] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proc. of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, 1329–1341.

[51] Jake Wharton. 2016. ButterKnife: Bind Android views and callbacks to fields and methods. https://github.com/JakeWharton/butterknife.

[52] Yichen Xie and Alex Aiken. 2007. Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability. *ACM Trans. Program. Lang. Syst.* 29, 3, Article 16 (May 2007).

[53] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proc. of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, 116–127.