Enabling Automatic Protocol Behavior Analysis for Android Applications

Jeongmin Kim* Hyunwoo Choi* Hun Namkung Woohyun Choi
Byungkwon Choi Hyunwook Hong Yongdae Kim Jonghyup Lee† Dongsu Han
KAIST †Gachon University

ABSTRACT

Understanding app-specific behavior is important for network operation and management. However, it is often difficult because it requires an in-depth application-layer protocol analysis due to the common use of HTTP(S) and standard data representations (e.g., JSON). This paper presents Extractocol, the first system to offer an automatic and comprehensive analysis of application protocol behaviors for Android applications. Extractocol only uses Android application binary as input and accurately reconstructs HTTP transactions (request-response pairs) and identifies their message formats and relationships using binary analysis. Our evaluation and in-depth case studies on closed-source and open-source apps demonstrate that Extractocol accurately reconstructs network message formats and characterizes network-related application behaviors.

1 Introduction

Android app is an important class of today's Internet applications that generate roughly 40-50% of mobile Web and app traffic [21]. More than 1.4 million Android apps are offered through Google's open market [18], and tens of thousands of new apps are added every month. However, very little information is known about Android application protocol behaviors because they predominantly use proprietary protocols on top of HTTP(S) [41, 48, 64, 77]. The problem is further exacerbated by the popular use of common data representation, such as JSON and XML. As a result, analyzing application protocol behaviors for Android applications requires an in-depth characterization of application-level payload for each individual application.

Understanding the application behavior within the network is invaluable in providing value-added services, such as application acceleration [4, 35] and dynamic caching [11]. It has been a significant interest for a number of applications, including testing protocol implementations [70], analyzing malware [75], and replaying application dialogs [31, 39, 68].

However, protocol behavior analysis is painstakingly manual, requiring tremendous amount of human effort [32, 38] in reverse engineering. On Android, the popular use of and default support for code obfuscation tools, such as ProGuard, make this even more difficult [24, 47].

This paper presents the first comprehensive protocol analysis framework that automatically extracts protocol behaviors, formats, and message signatures. Providing automatic analysis for Android apps, however, introduces two unique challenges. First, while application binary is readily available through the open market, only the client program is available. Typically the server binary, protocol documentation, and the source code are unavailable. Therefore, we must solely rely on the client program, unlike other approaches that use both the server and client binary [32, 68] or even the source code [70]. Second, it must provide high coverage and accurately infer the message format and track dependency relationships between messages—e.g., an authentication token to be used for subsequent requests may be embedded in a prior login response, or a URI of an object may be embedded in a prior response.

To ensure high coverage and accuracy, we apply static binary analysis, using the Android application binary as input. Our insight is to track the code that generates or processes network messages. Our system, Extractocol, infers dependency relationships of objects that flow in from and flow out to the network buffer using static binary analysis. In particular, it extracts parts of application code (i.e., program slices) that either generate requests or parse response messages. It then internally reconstructs the dependencies between these objects. Finally, it applies a careful semantic analysis to extract message formats and signatures from the target program.

Extractocol outputs signatures for each request/response (including URI, query string, request method, header, and body). By pairing a request with its corresponding response, it accurately reconstructs HTTP transactions. We primarily use regular expression to represent the signatures. However, Extractocol internally maintains a tree representation of a signature, allowing us to represent signature in other forms, such as Document Type Definition (DTD) for XML

*The first two authors contributed equally to this work.

CoNEXT '16, December 12 - 15, 2016, Irvine, CA, USA

 ISBN 978-1-4503-4292-6/16/12...$15.00

DOI: http://dx.doi.org/10.1145/2999572.2999596

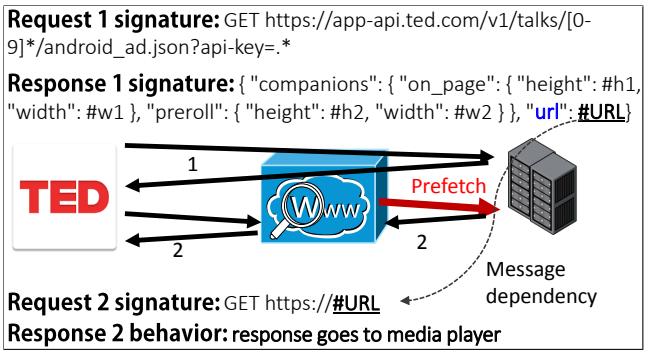

Figure 1: Application acceleration example

and JSON schema for JSON bodies. Extractocol also infers dependencies between transactions by tracking which fields in a request message (e.g., an authentication token) come from an earlier response. Finally, it is able to track how the network originated data is consumed within the Android app (e.g., network data is fed into a video player).

In summary, this paper makes three key contributions:

- **Automatic protocol analysis:** We present the first comprehensive protocol analysis framework for Android applications that is capable of extracting protocol behaviors, formats, and signatures.
- **Novel application of static taint analysis for inferring protocol messages and their relationships:** Extractocol reconstructs request/response pairs, infers fine-grained dependencies between protocol messages, and obtains accurate protocol message formats.
- **System prototype and evaluation:** Our in-depth analysis on open-source and closed-source apps shows that it provides a rich and comprehensive characterization.

2 Motivation and Approach

Applications of protocol analysis: Understanding the app behavior has significant implications in networking. Apart from its *intrinsic* value, it enables the network to provide new application-aware services, often in conjunction with other networking technologies. We provide a few examples of how our protocol analysis can potentially be applied to enable new services.

First, it enables application-aware treatment in the network. Using the request signatures and fine-gained message dependencies, one can intelligently prefetch content, which is one of the key building blocks for application acceleration [4, 7, 11]. Figure 1 shows part of the analysis result of Extractocol for TED Android app. When a talk is requested (request 1), its response contains the URL of an advertisement video. The URL is then requested and the response goes to the video player. Because Extractocol automatically identifies this, one can generate a prefetcher that prefetches advertisements. Also, with the identification of protocol fields and message formats, app-specific dynamic caching can also be automated to accelerate particular applications [11, 16]. Today, the development of dynamic caching proxies is done manually on a per-app basis [16] because it requires the knowledge of application semantics (e.g., which request parameter is dynamically generated) to determine which content is cacheable [11]. A preliminary study has demonstrated the effectiveness of this approach [35] by using the information from tool, Extractocol. The practice is referred to as dynamic site acceleration. For example, Akamai offers a dynamic site acceleration solution [5] where Web applications are accelerated using proxies near the end user, obtains TLS certificate and private key for the service, and serve prefetched dynamic content from the proxy. We believe developing application proxies [35] can be automated and Extractocol is the first step towards this.

In addition, our framework tracks how network data is consumed (e.g., media player) and where the network bound data is originated from (e.g., microphone). This information can be valuable to the network in enhancing the quality of user experience. For example, if the network knows that the response message is streamed into a media player, rather than to a file, it can treat the traffic as such. Similarly, if the app streams data from the microphone or camera, we might infer that the traffic is of high priority or latency-sensitive. Currently, QoS configuration (e.g., preferential traffic treatment) is largely done manually and coarse grained (e.g., on a per-port or per-flow basis). We believe that the understanding of application behavior combined with advances in software-defined networking [44, 71, 88] and visibility of encrypted traffic [6, 13, 14, 67] can enable new dimensions of application-aware networking.

Second, the knowledge on protocol behavior can be used to automate protocol testing. Testing the protocol behavior is often cumbersome and tedious because it requires generating protocol messages exhaustively and also protocol messages often have orderings due to their dependencies. Application protocol analysis can potentially automate this process by generating messages exhaustively while following the dependency between message exchanges.

Third, it can provide useful information for detecting security and privacy issues. For example, taint analysis has long been used for detecting sensitive information leakage [46, 52, 86]. In addition, our tool can be used to finger-print the protocol behaivor of apps, which may be helpful in detecting malicious app behaviors or app repackaging. For example, one approach to malware detection is to match signatures represented as binary patterns or plain strings. However, malwares can easily evade the mechanism by reordering their instructions or functions [10]. In contrast, Extractocol extracts network behavior including network signatures and dependencies between HTTP transactions. This can help detect malware variants in a more robust fashion (unless malwares change their network behaviors significantly).

Goal: Our goal is to provide a comprehensive analysis of each individual application, rather than a large-scale analysis. Note, for the applications above, providing accurate and in-depth analysis with wide coverage is critical. Our system primarily focuses on extracting the application layer interaction that happens over HTTP(S) because most Android apps use HTTP as their primary protocol. Extractocol strives to provide a comprehensive characterization in many regards: it 1) captures all HTTP interactions and reconstructs each transaction; 2) infers relationships and fine-grained dependencies

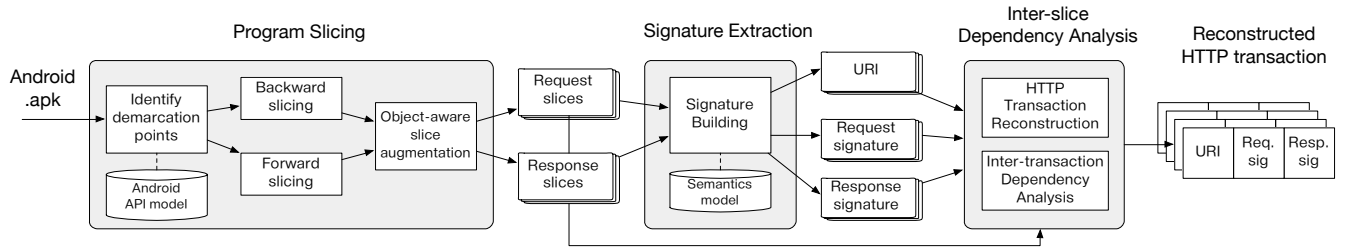

Figure 2: Design overview of Extractocol

between messages; and 3) enables analysis even when apps use HTTPS, which is especially useful for enterprise security applications that decrypt SSL/TLS traffic for enhanced visibility [13, 14, 67].

An HTTP transaction consists of URI, request data (header, mime-type and body), request method, and response data. Our system outputs URI, text, and body signatures using regular expressions.

Approach: Our key insight in protocol analysis is to track all objects that eventually go out to or flow in from the network buffers. To this end, Extractocol identifies and analyzes the program slices that generate/process protocol messages using data dependency analysis. In the process, we solve a number of core challenges for reconstructing program semantics and protocol-related data flows.

The primary contribution is that it is the first system that comprehensively extracts protocol messages and behaviors from Android applications. Our secondary contribution is its novel application of static taint analysis. In particular, we apply taint analysis in three different ways to obtain data dependencies that result from protocol processing. [1] For tracking protocol-related data flow, we extend FlowDroid [27] that provides flow-, context-sensitive, and inter-procedural data flow analysis on Android apps.

3 Extractocol Design

To achieve the end goal, Extractocol performs three tasks: it 1) creates program slices that capture network interaction; 2) incorporates semantics analysis with data dependency analysis to reconstruct message formats and signatures; and 3) identifies fine-grained dependencies between protocol messages by discovering inter-slice dependencies. Figure 2 illustrates the three components.

Network-aware program slicing: A typical program contains many instructions other than protocol processing. Thus, Extractocol pre-processes the APK to extract program slices only related to protocol processing. The goal of this step is to output program slices that generate HTTP requests and process responses. Extractocol extracts all program slices that encompass the objects that either go out to or flow in from the network. We name the out-bound data flow as *request slice* because it captures the code and objects for constructing a request, and the in-bound flow as *response slice* because it captures the code and objects used for processing a response. To obtain these slices, Extractocol employs novel bidirectional taint analysis (§3.1).

Signature extraction: The second phase takes the request/response slices as input and generates message signatures. Since the program slice captures all objects and operations that generate a request or process a response, it encodes all necessary information to extract their signatures. For signature extraction, Extractocol utilizes semantic models for commonly used Android and Java APIs for HTTP processing. It then outputs the request method and signatures for request URIs and request/response headers and bodies (§3.2).

Message dependency analysis: Finally, Extractocol reconstructs a complete transaction by pairing a request URI with its corresponding response. It also infers the relationship between HTTP transactions. In particular, it infers which part of request URI or body is potentially derived from prior responses. The key idea is to identify inter-slice relationships between the request and response slices. For this, Extractocol performs novel inter-slice data flow analysis and addresses a number of issues in handling subtle, but complex inter-slice dependencies that arise due to code reuse (§3.3).

3.1 Network-aware Program Slicing

Extracting program slices that process protocol messages requires keeping track of all operations on data objects that are network I/O bound.[2] Extractocol applies static taint analysis to track network-bound information flow. Unlike static taint analysis whose primary goal is to determine the existence of data flow from taint sources to sinks, Extractocol must track *all operations* on network-bound objects for reconstructing message signatures. Omitting even a single statement that operates on these objects would result in an inaccurate signature. We explain how Extractocol achieves this efficiency.

Demarcation points and bi-directional slicing: Because any object can turn out to be network I/O bound, a naive use of taint analysis would require keeping track of *every* object and its data flow, which is computationally too expensive. Another strawman approach is to carefully select the taint source and sink objects. For example, taint sink can be network access objects (e.g., `org.apache.http`), and source can be JSON, XML, and URI objects. However, this approach is not general enough because generic objects (e.g., array) can turn out to be network bound.

Instead, our main idea is to start from network access methods and taint network buffers. For example, if we taint

[1] Taint analysis is used for bi-directional slicing, inter-slice dependency analysis, and handling asynchronous events as we describe in §3.

[2] We refer to objects as network I/O bound if they either originate from network messages or eventually go out to the network.

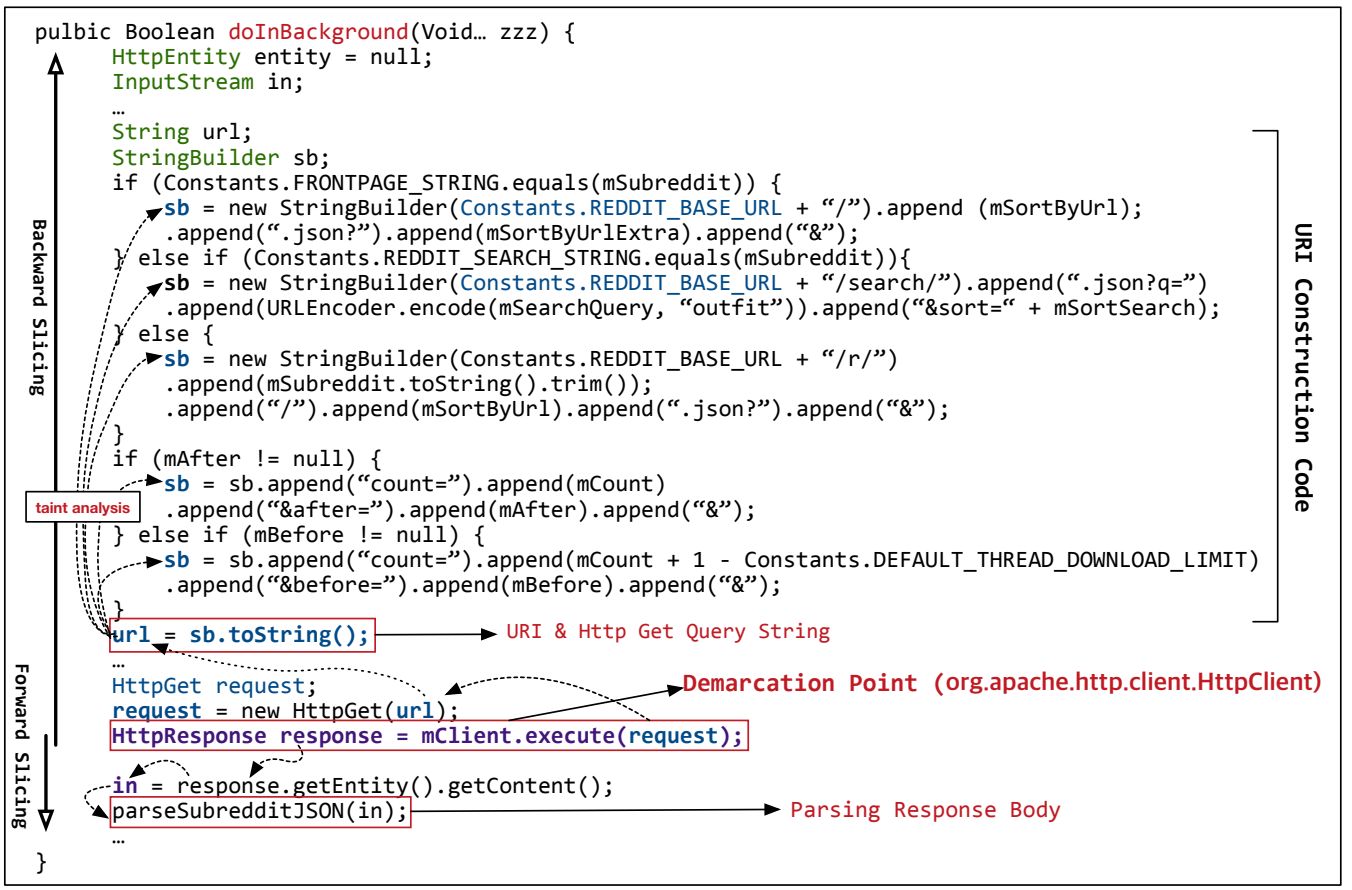

Figure 3: Reddit Client (Diode)'s Request & Response slice example

socket buffers, such as `socket.getOutputStream()` and `socket.getInputStream()`, and perform taint propagation, we would be able to track all objects that read from input or write to output socket buffers. Extractocol applies this idea to HTTP processing. We taint HTTP request/response objects used for HTTP interactions, such as `HttpRequest` and `HttpResponse`, as input and output buffers.

Figure 3 shows a code snippet from a reddit client. HTTP access functions, such as `HttpClient.execute()`, take in a `HttpRequest` object and return a `HttpResponse` object. From these statements, Extractocol performs bi-directional (backward and forward) taint propagation, using method's host object, parameters, and return object as the initial tainted objects. The intuition is that forward taint propagation from these functions would track objects that originate from the network (e.g., derived from `HttpResponse`) and backward tainting would track objects that write to the output buffer, such as `HttpRequest`. We refer to such HTTP access functions as *demarcation points (DPs)* because they separate the forward and backward program slices.

As shown in Figure 3, forward taint propagation reveals the data dependency for objects related to response message processing, and backward tainting relates objects that make up the URI, request method, and body. Note, the problem is now reduced to selecting DPs from well-defined Android and Java APIs, which is much more tractable than tracking all objects and much more accurate than heuristically selecting network-bound objects. For this, our semantic model contains information on commonly used HTTP libraries. To help program analysis, the semantic model encodes the semantics of commonly used Android libraries for HTTP processing.

Bi-directional propagation: For high accuracy, the program slices must contain *all operations* related to network-bound objects. To construct such a slice, Extractocol performs open-ended taint propagation to add all statements that include tainted objects into the program slice. Forward taint propagation is handled by FlowDroid's default tainting rules. For backward propagation, however, we flip the edge direction of the control flow graph to inspect the statements in the reverse order. We then apply inverted taint propagation rules—we swap the premise and conclusion of the rules for intra-procedural flows and reverse the taint rules for call and return flows to follow the inter-procedure graph in a reverse order. Namely, a tainted LHS (left-hand side) taints RHS (right-hand side) in an assignment statement, and the taint information of callee's arguments is propagated to caller's arguments. We trace the tainted objects until it has no more objects to propagate. In backward taint propagation, an object is untainted at its definition. In forward taint propagation, objects destructed at the end of procedures are untainted.

Object-aware augmentation: Although, the resulting forward slice reveals data flow for response processing, one limitation is that it may not be self-contained. For example, if an object used in a forward slice is initialized before

the demarcation point (DP), the slice does not contain the initialization parameters.

Extractocol augments forward slices with the complete context of objects contained within. It identifies all objects within each forward (response) slice and inspects all backward slices that share the same DP. For each statement in the backward (request) slice, Extractocol checks whether it has direct dependency with objects in the forward slice. If so, we include the statement. We repeat the process until no statements are added. Also, we handle references to resource objects, such as Android.R, whose values are stored user-defined files in the APK (e.g., `res/values/strings.xml`).

3.2 Signature Extraction

This phase takes each request/response slice as input and extracts their formats and signatures and compiles them into regular expressions. It is logically divided into two steps: 1) Extractocol identifies the objects corresponding to URI, request body, and response body through a semantic analysis; for each of the three main objects, it obtains a sub-slice that encompasses statements from the initialization of objects that influence the main object up to their final use. 2) Extractocol actually builds a signature for the sub-slice using semantic models of commonly used Android and Java APIs.

Semantic model: Both steps require program semantic analysis. For this, Extractocol uses semantic models for a set of Android and Java APIs that are commonly used for HTTP protocol processing. The model captures the semantics of each API's operations and its parameters. We model methods and interfaces commonly used for network/HTTP message processing. In particular, we model high-level Java and Android APIs, such as `org.apache.http`, `android.net.http`, `com.android.volley`, `google-http-java-client`, and `java.net`, and popular third-party HTTP libraries [17, 65], such as `OkHttp`, `Retrofit`, `Beefr amework` and `rx.android`, for identifying HTTP-related objects, eight commonly used JSON and XML libraries [1], basic containers, such as Array and List, basic Java/Android methods often used for protocol processing, and string/byte manipulation APIs. We also support reflection-based JSON libraries such as Jackson, gson, and retrofit. We find that our model is sufficient for modeling many real-world applications (see §5). To be extensible, we also provide an easy plugin for adding new API semantics. While we currently employ manually derived API models, we believe that Extractocol can be extended to automatically infer the semantics of high-level APIs, as their implementations commonly rely on low-level string manipulations or socket API calls.

1) Semantics-aware object identification: Using the semantic models, Extractocol identifies the URI object, request and response body, and the request method. Extractocol then logically separates the slices by creating a dependency graph for each of the three objects (URI, request, and response body), using the information obtained from program slicing. This process refines the slices for signature building and logically separates the request slice into two sub-slices: one for request body and the other for URI generation. Note the URI slice contains not only the URI, but also the request method and additional HTTP headers that application uses.

sig_pat	::=	$term \mid \texttt{concat}(term, term)$
		$\mid \texttt{rep}\{term\} \mid term \vee term$
term	::=	$constant \mid struct\_str \mid \texttt{unknown}$
struct_str	::=	$\texttt{json}\ (\ obj\ ) \mid \texttt{xml}\ (\ obj\ )$
obj	::=	key_value^*
key_value	::=	$(\ key, value\ )$
key	::=	*constant*
value	::=	$constant \mid obj \mid array$
constant	::=	$\texttt{num}$ integer $\mid \texttt{str}$ string
array	::=	$value^*$

Figure 4: An intermediate language for signature

2) Signature building: Extractocol inspects the three sub-slices to construct their signatures, using the same semantic model. It processes each statement in order and maintains data structures to reconstruct data operations encoded in the slice. For string objects, such as request URIs, query strings, and text bodies, Extractocol maintains a simple data structure that keeps track of the string literals and objects being written to the string object and their relative offsets. For JSON and XML objects, Extractocol maintains a tree data structure.

For each tainted object, it maintains a signature in the signature database. When it encounters a statement that updates tainted objects, it keeps track of the string literals and objects being written to the tainted object and updates their signatures according to the semantics (e.g., string append or JSON put). To aid this process, we use an intermediate language to represent the signature whose simplified form is summarized in Figure 4. The language encodes objects and operations, such as string literals, structured strings, concatenation, repetition, disjunctions, and array. Next, we describe the signature building algorithm in more detail.

Given the program slices, Extractocol inspects each procedure following the inter-procedural data flow encoded in the program slice. *Within* a procedure, it performs flow-sensitive analysis to accurately capture the signatures from all program flows as it may contain branches. For this, we design a new algorithm that efficiently traverses the intra-procedural control flow. Traditional data-flow analysis, such as the worklist algorithm, uses iterative methods to obtain a fixed-point solution. For every change, it revisits the statements in the basic blocks that may be influenced by the change. However, it is known to be very slow and has scalability issues. To be more scalable, we leverage the fact that our objective is to extract signatures for protocol messages and the signatures can be conservatively expressed in our language. We process the statements in basic blocks in topological order of the intra-procedural control flow graph and build the signature database that maps a variable to its signature for each basic block. At confluence points, we merge the signature database from each flow, in the following way. If all the instances of a variable are well-defined (not `unknown`) and the confluence point is not a loop header or latch, Extractocol merges signatures for the variable with logical disjunction ($\vee$). If the confluence point is a loop header or latch, Extractocol

identifies the loop variant part of string objects and denotes it uses `rep` to mark the part can be repeated in the signature.

Finally, Extractocol converts the signature for URI, request, and response objects into regular expressions. The regex format of a variable object is derived from its type (e.g., [0-9]+ for integer variables and .* for string variables). Repetitions (`rep`) and disjunctions (`V`) are respectively converted into the Kleene star (*) and | in regular expressions. For JSON and XML objects, the result are of tree structure whose leaves are string literals or numbers.

Example: Figure 3 shows an example code for Diode, a popular open-source browser for Reddit, on Google Play[3]. Extractocol's network-aware program slicing effectively identifies the request/response slices. The resulting slices only contain 6.3% of all code, making the signature building process very efficient.

Extractocol then builds signatures on the slices. From `HttpClient.execute()` (demarcation point in Figure 3), it identifies a request object, '*request*.' It also recognizes the request method, `GET`, from the object's initialization (i.e., *request* is an *HttpGet* object). Traversing basic blocks, Extractocol inspects each statement of the request slice and extracts nine URIs. It employs API semantic model for `StringBuilder`, `org.apache.http.message`, and `List` to generate signatures for the URIs. Finally, our flow-sensitive signature building process outputs a regular expression that combines all nine request URI patterns in Figure 3, one of which is `http://www.reddit.com/search/.json?q=(.*)&sort=(.*)`.

3.3 Message Dependency Analysis

Extractocol infers the relationships between messages by identifying inter-slice dependencies.

Request-response pairing: We now pair the HTTP request and response messages derived from Extractocol. We use information flow analysis to identify the response that depends on a given request. For this, Extractocol uses URI slices as taint source and response slice as sink and performs taint propagation described in §3.1. However, it is not trivial because of code reuse. When multiple requests and responses share a *common* demarcation point, standard information flow analysis results in a failure; it identifies multiple responses for a single request URI. Figure 5 illustrates such an example.

Two requests and two response slices exist for transactions A and B. Starting from a common method, their paths diverge into `requestA()` and `requestB()`, but soon reconverge to share a common demarcation point in `common2()`. Each box represents a method, and notable code segments are labeled with numbers from 1 to 6. Code segments 1 and 6 respectively mark the beginning of a request and end of response processing for both A and B. As shown in the figure, using request slices (which includes 1) as the taint source and response slices (which includes 6) as sink does not identify a one-to-one relationship. Information flow analysis

[3]Although Extractocol analyzes Dalvik bytecode in an intermediate language, we show its source code for illustration purpose.

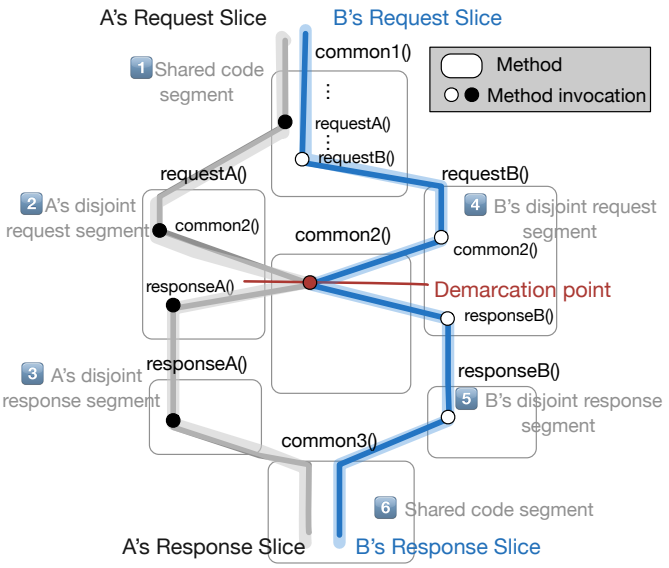

Figure 5: Extractocol locates disjoint code segments, between which only a single path exists.

would discover paths to both A's and B's responses from A's request (i.e., two paths exist from 1 to 6). Extractocol addresses this problem using a simple intuition: If all request/response slices are disjoint, one-to-one relationship would hold between them. Motivated by this, Extractocol uses disjoint sub-slices (e.g., segments 2, 3, 4, and 5) as input by preprocessing the slices. The information flow analysis then discovers a path from 2 to 3 and another from 4 to 5. Thus, we can pair A's request with A's response slice and not with B's response slice. However, pairing may not always be one-to-one in general as there might be a common response handler for multiple requests.

Inter-transactional dependency: Extractocol also identifies fine-grained dependencies by inferring whether objects that are derived from a response is used to construct another request. For this, we identify all objects modified/set as a result of response processing (i.e., response-originated objects) during signature extraction. We also track all objects that make up a request (i.e., request-originating objects). Extractocol infers potential dependency by checking whether the set of response-originated objects overlap with the set of request-originating objects. If they do, we conclude the two transactions have potential dependencies. To ensure object and field sensitivity, we perform object alias analysis by locating common initialization points between the two objects that might overlap. We also track dependencies in field granularity. Extractocol finally outputs which request fields originate from which response fields, which is useful for inferring the protocol common usage and future requests.

3.4 Handling Practical Issues and Challenges

To ensure accurate analysis for real-world apps, Extractocol solves practical challenges in Android program analysis.

Asynchronous events: Asynchronous event handling is very common in Android programming. For example, in our dataset, a weather notification app sets its location inside a callback invoked by a location service. It constructs a part of

query string that contains city names and GPS locations into a heap object. Later, another event, such as a user click, actually reads the object to generate an HTTP request. In another example, a server message sets a sub-URI and a timer event triggers a request to that URI. Although, this creates an implicit data flow, it is difficult to infer the ordering between two asynchronous events using static analysis [27]. FlowDroid, in particular, assumes an arbitrarily ordering of these events. This results in a failure to identify the dependencies across these events. In the example above, the dependency between the first (location service) and second event (user click) is lost. Thus, the part of query string created by the location service cannot be identified. Note, dynamic analysis does not solve the problem either because it is difficult to generate all asynchronous events (e.g., server push and timers).

To address the problem, Extractocol tracks all objects that make up a request. For each such object, we identify statements and methods that modify the object (e.g., setter for the object). We then perform backward taint propagation (§3.1) from these statements. This identifies the callers of these functions and the statements that are used to construct the objects in the first place; e.g., it identifies the callback function for location service that constructs part of the URI in the weather app example. In this fashion, we effectively identify implicit data flows that potentially impact the request. Our experience shows that, for closed-source apps with rich UI, this dramatically improves the signature accuracy.

Implicit call flow: Network programming in Android often involves using thread libraries such as `AsyncTask` [3], which introduce implicit call flows. However, existing static taint analysis tools often do not cover them. A recent study, EDGEMINER [33], identified 19,647 callbacks, many of which FlowDroid does not handle. In fact, discovering all implicit callbacks in Android is an active area of research, and a number of studies [33, 69] are devoted to addressing the issue. To enhance the coverage of Extractocol, we add support for many popular implicit callbacks commonly observed in network operation and HTTP libraries [3, 17, 65], such as `AsyncTask`, `volley`, and `retrofit`.

Handling obfuscated libraries: Obfuscation is commonly observed in popular real-world apps. A recent study has shown that 15% of apps are obfuscated [79]. Many tools, including Proguard, rename identifiers with semantically obscure names to make reverse engineering more difficult. Extractocol handles obfuscated application code without any modifications because identifier renaming does not affect its operation. However, when library code included in our semantic model is obfuscated, it becomes more challenging because it requires de-obfuscation. To handle this case, we pre-process the code to generate a map between the obfuscated identifier and the original one. For this, we compare the signatures of the method contained in our semantic model to identify the class and method that has the most similar signature patterns. When there are multiple methods with the same signature, we look at the decompiled code and look for similarity. We find that many real-world apps do not obfuscate library codes, even when their own code is obfuscated. Note, similar techniques have been used in other studies for identifying obfuscated ad libraries in Android apps [62]. Alternatively, one can also use advanced de-obfuscation tools [9, 12] that also handle encrypted strings.

4 Implementation

Our implementation consists of two modules: the data flow analysis module and the signature generation module. The data flow analysis module consists of 2,953 lines of Java code for program slicing and identifying inter-slice dependencies. For this, we extend FlowDroid, a static taint analysis framework, that identifies whether a path exists from a source of sensitive information to a taint sink. It builds upon many existing frameworks such as: Soot, a static analysis framework that provides the Jimple intermediate representation for Java and a call graph analysis framework [60]; Dexpler [28] that converts Dalvik bytecode to Jimple; and IFDS [73] that provides inter-procedural data flow analysis. We modify FlowDroid to find demarcation points and perform bi-directional tainting when necessary. The current implementation of Extractocol uses 39 demarcation points from 16 classes and popular http libraries [17, 65], including `org.apache.http`, `android.net.http`, `android.volley`, `java.net`, `android.media`, `retrofit`, `BeeFramework`, and `ok http`. We also support dependencies that occur between implicit call flows of thread libraries, such as `AsyncTask`, `Volley`, `rx.android`, `FutureTask`, and `Beeframework`.

The semantic analysis module consists of 8935 lines of Java code. It takes as input an abstract syntax tree of program slices, represented by modified Soot classes. Extractocol operates at Jimple/Shimple code level, instead of the Dalvik bytecode.[4] For the API semantic model, we build-in a number of low-level string APIs and generic data types, including `List`, `Array`, and `HashMap`. Our model also supports the HTTP(S)-related libraries and eight XML and JSON APIs, including `org.json`, `com.google.gson`, `org.xml`, and `com.fasterxml.jackson`, and supports reflection-based nested json serialization.

Discussions and limitations: Extractocol addresses many first-order issues in analyzing HTTP-based application protocols on Android. However, it currently does not handle direct use of `java.net.socket`, automatic modeling of high-level API semantics, binary protocols over HTTP, Android intents, and native binary. Some of them are implemention issues, and Extractocol can be extended to support most of them. Direct use of socket can be handled by modeling socket APIs because Extractocol already parses text-based protocols. API semantics can be automatically inferred by inspecting its code, similar to how we reconstruct signatures from a model of low-level string APIs. Parsing binary protocol requires modeling byte operations. Intents can be also handled by modeling the implicit control flow it introduces, similar to how we handle threads. Supporting native code and JNI is out-of-scope, but more challenging because static analysis

[4] Jimple is a popular intermediate language based on three address code (3AC) often used for bytecode optimization. Shimple only uses the static single assignment (SSA) form.

App	Protocol	Request URI				Request Body/	Response		#Pair
		GET	POST	PUT	DELETE	Query string	JSON	XML	
		(**Extractocol** / Man. UI fuzzing / Source Code)							
Adblock Plus	HTTPS	**2** / 2 / 2	**1** / 1 / 1	-	-	**1** / 1 / 1	**-**	**1** / 1 / 1	1
AnarXiv	HTTP	**2** / 2 / 2	**-**	-	-	-	**-**	**2** / 2 / 2	2
blippex	HTTPS	**1** / 1 / 1	**-**	-	-	**-**	**1** / 1 / 1	**-**	1
Diaspora WebClient	HTTP	**1** / 1 / 1	**-**	-	-	**-**	**1** / 1 / 1	**-**	1
Diode	HTTP(S)	**24** / 24 / 24	**-**	-	-	**-**	**2** / 2 / 2	**-**	5
iFixIt	HTTP	**15** / 15 / 15	**7** / 7 / 7	-	-	**3** / 3 / 3	**14** / 14 / 14	**-**	14
Lightning	HTTP(S)	**2** / 2 / 2	**-**	-	-	**-**	**-**	**1** / 1 / 1	1
qBittorrent	HTTP	**3** / 3 / 2	**13** / 13 / 2	-	-	**13** / 13 / 13	**3** / 3 / 3	**-**	3
radio reddit	HTTP(S)	**3** / 3 / 3	**3** / 3 / 3	-	-	**3** / 3 / 3	**4** / 4 / 4	**-**	4
Reddinator	HTTP(S)	**3** / 3 / 3	**3** / 3 / 3	-	-	**-**	**6** / 6 / 6	**-**	6
Twister	HTTP	**-**	**11** / 11 / 11	-	-	**11** / 11 / 11	**8** / 8 / 8	**-**	8
TZM	HTTPS	**2** / 2 / 2	**-**	-	-	**-**	**1** / 1 / 1	**-**	1
Wallabag	HTTP	**1** / 1 / 1	**-**	-	-	**-**	**-**	**1** / 1 / 1	1
Weather Notification	HTTP	**2** / 2 / 2	**-**	-	-	**-**	**-**	**2** / 2 / 2	2
		(**Extractocol** / Man. UI fuzzing / Auto UI fuzzing)							
5miles	HTTPS	**24** / 25 / 0	**51** / 12 / 0	**-**	**-**	**16** / 6 / 0	**16** / 8 / 0	-	71
AC App for Android	HTTP(S)	**9** / 9 / 7	**15** / 15 / 5	**-**	**-**	**15** / 15 / 15	**23** / 23 / 23	-	23
AOL: Mail, News & Video	HTTP	**9** / 9 / 6	-	**-**	**-**	**-**	**9** / 9 / 9	-	9
AccuWeather	HTTP	**15** / 15 / 0	**3** / 3 / 0	**-**	**-**	**3** / 3 / 3	**16** /16 / 16	-	16
Buzzfeed	HTTP(S)	**16** / 5 / 5	**12** / 5 / 1	**-**	**-**	**28** / 5 / 5	**6** / 5 / 5	-	27
Flipboard	HTTPS	**23** / 24 / 0	**41** / 13 / 0	**-**	**-**	**28** / 13 / 0	**8** / 7 / 0	-	63
GEEK	HTTPS	**0** / 1 / 0	**97** / 48 / 18	**-**	**-**	**41** / 48 / 18	**11** / 27 / 18	-	97
KAYAK	HTTPS	**39** / 39 / 15	**7** / 7 / 5	**-**	**-**	**7** / 7 / 7	**6** / 6 / 6	-	6
Letgo	HTTPS	**38** / 32 / 10	**10** / 14 / 2	**2** / 2 / 0	**3** / 0 / 0	**20** / 14 / 3	**18** / 13 / 6	-	40
LinkedIn	HTTPS	**38** / 42 / 16	**49** / 17 / 8	**0** / 3 / 0	**-**	**46** / 17 / 14	**47** / 21 / 14	-	85
Lucktastic	HTTPS	**16** / 2 / 0	**9** / 15 / 0	**2** / 0 / 0	**4** / 0 / 0	**5** / 15 / 0	**19** / 14 / 0	-	31
MusicDownloader	HTTPS	**3** / 10 / 0	**0** / 1 / 0	**-**	**-**	**0** / 1 / 0	**4** / 7 / 0	-	2
Offerup	HTTPS	**33** / 20 / 0	**23** / 21 / 0	**8** / 1 / 0	**3** / 0 / 0	**12** / 21 / 0	**25** / 16 / 0	-	63
Pandora Radio	HTTP(S)	**7** / 0 / 0	**53** / 20 / 2	**-**	**-**	**53** / 20 / 2	**26** / 16 / 2	-	60
Pinterest	HTTPS	**60** / 62 / 26	**36** / 19 / 16	**32** / 8 / 3	**20** / 10 / 2	**88** / 19 /36	**236** / 58 / 46	-	148
TED	HTTP(S)	**16** / 16 / 10	**2** / 2 / 1	**-**	**-**	**2** / 2 / 2	**10** / 10 / 10	-	10
Tophatter	HTTPS	**33** / 24 / 0	**32** / 14 / 0	**1** / 0 / 0	**4** / 1 / 0	**18** / 14 / 0	**32** / 11 / 0	-	62
Tumblr	HTTPS	**12** / 13 / 15	**8** / 5 / 5	**-**	**1** / 1 / 0	**5** / 5 / 15	**14** / 2 / 14	-	20
WatchESPN	HTTP	**33** / 33 / 17	-	**-**	**-**	**-**	**32** / 32 / 32	-	32
Wish Local	HTTPS	**0** / 1 / 0	**106** / 48 / 21	-	**-**	**15** / 15 / 21	**28** / 13 / 21	-	106

Table 1: Signatures identified for open-source and closed-source apps (gray box)

on native code is not mature enough for extracting protocols from realistic native binary [82]. One approach is to use dynamic analysis on native code [2, 72] to reconstruct the semantics of JNI.

Finally, our current implementation does not track dependencies across multiple chains of asynchronous events, but only detects dependencies across on hop. One can perform multiple iterations until it does not discover new dependencies for better accuracy and wider coverage.

5 Evaluation

Our goal is to provide a comprehensive analysis of individual applications, rather than a large-scale analysis. Using in-depth case studies of 14 open-source and 20 closed-source apps, we demonstrate that the direct output of Extractocol offers an in-depth analysis of Android applications.

The open source apps were obtained from the open source app repository (F-Droid) [8] (twelve of them are also available on Google Play). We select very popular closed-source apps with 1 million+ downloads that primarily use HTTP and JSON/XML. We exclude apps that primarily use binary protocol or use intent based communication to generate protocol messages because Extractocol currently does not support them (§5.3). Most of them are within top 100 in the Google Play applications category [15].

Our evaluation answers three key questions:

1. *Does it provide accurate signatures with wide coverage for each application?* (§5.1)
2. *Does it effectively characterize app behaviors?* (§5.2)
3. *Can it reverse-engineer (private) REST APIs?* (§5.3)

5.1 Validation of Protocol Analysis

For open source apps, we disable the heuristics for handling asynchronous events, but enable it for closed-source apps. Note, many closed-source apps are obfuscated. For open source apps, we obfuscate their APKs using ProGuard [24] and verify that the same results hold as non-obfuscated APKs. Extractocol takes 4 minutes to analyze an open source app on average. For closed-source apps, the time varies widely from 11 minutes (for a small app) up to 3 hours (for a large app).

Criteria: A protocol analysis must provide high *coverage* by identifying as many HTTP request/response messages as

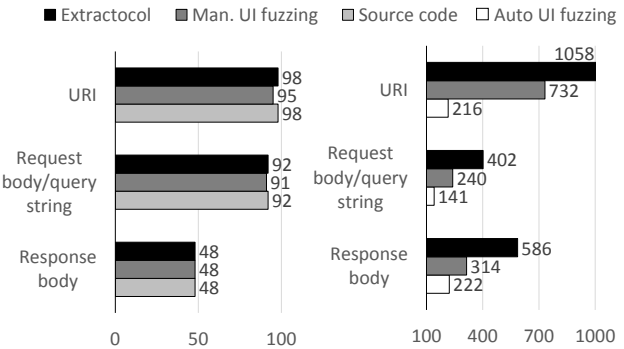

Figure 6: Open source (left) / closed-source (right) apps # signatures

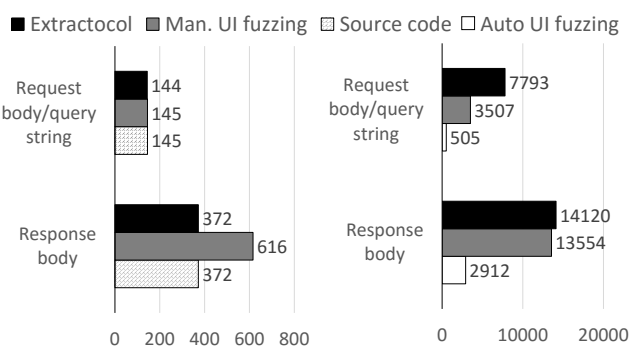

Figure 7: Open source (left) / closed-source (right) apps # keywords

possible. At the same time, its signatures must be *logically equivalent* to the operations encoded in the target program and generate a *valid* match on actual network traces. We verify these criteria as well as *signature quality*.

Coverage: We compare the result of Extractocol with that of manual fuzzing and automatic fuzzing for closed-source apps. For open source apps, instead of automatic fuzzing, we obtain the ground truth by carefully inspecting the *source code*. For all apps, we collect traffic traces of all HTTP(S) transactions using manual UI-fuzzing. Note this often requires manual interventions, such as signing up and logging in for services. For automatic fuzzing, we use PUMA, a UI-automation tool developed for efficient fuzzing.[5] We use man-in-the-middle proxies [19, 23] to capture and decrypt HTTPS messages. We then match the traffic traces with our regex signatures.

Table 1 shows the number of unique HTTP request/response for all apps (e.g., Adblock Plus has two GET and one POST request, out of which one generates an XML response body). For open source apps, the three numbers in each cell respectively show the results for Extractocol, manual UI fuzzing, source code analysis. For closed-source apps (gray), they show results for Extractocol, manual UI fuzzing, automatic UI fuzzing, respectively. Figure 6 shows the total number of unique URIs, request body/query strings, and response bodies identified by each methods.

Extractocol generates much more unique message signatures than manual and automatic fuzzing with PUMA. PUMA fails to recognize custom UI for a number of apps (Table 1) and stops to explore further. One can add PUMA scripts to handle custom UIs for each app. However, UI-fuzzing, including manual fuzzing, fundamentally falls short in providing wide coverage for two reasons. First, some messages are not triggered by human-generated events. For example, we find that some apps trigger APK update requests using timers. Second, some messages are only triggered by actual "actions" with side-effects, such as purchasing products or applying jobs to a company (e.g., LinkedIn). This makes it difficult to explore all cases without a fake server that acts just like the original. In fact, many of the top apps (e.g., 5miles, Letgo, Tophatter, Offerup, Wish Local) are e-commerce/shopping apps where actions, such as payment, delivery, selling and purchasing products, generate HTTP interactions. Upon inspecting the signatures that we did not have traffic traces for, we find that they are related to processing payments, registering new products for sale, viewing products the user has sold/purchased, rating products one purchased, and tracking deliveries.

However, for some apps, Extractocol misses some messages. This is due to two limitations discussed in §4: 1) some messages are triggered by intents; and 2) some apps have multiple chains of dependencies across asynchronous events. We identified that some ad and analytics libraries extensively use multiple asynchronous events and intent. Most of the missed messages stem from the libraries. For example, Luctastic uses various ad and analytics libraries such as chartboost, tapjoy, tap-nexus.appspot, vungle, amplitude and adjust.

Finally, Extractocol pairs responses that have bodies processed by the apps, and it identified 971 HTTP (request URI-response body) pairs. We successfully verified that the signatures and the request-response pairs successfully match with the traffic traces obtained from fuzzing (if present).

Signature validity: For all signatures that have corresponding traffic traces from manual fuzzing, we match them using a regex matcher. All such signatures generated a valid match with the actual traffic trace. In sum, Extractocol provides wide coverage and generates correct signatures.

Logical equivalence: For open-source apps, we verify whether our regex signature and pairings are logically equivalent to the source code. We manually inspect the source code of all 14 open source apps and identify the slices that generate URI, request query and body and that parse the response. We then carefully inspect if the signature and pairing is logically equivalent to the program slice. For example, for request URI, we manually verify if our URI signature accurately represents the URI created by the program slice without unneccessary strings. For a JSON response, we inspect if our signature accurately represents the JSON tree that the response slice is trying to parse. We found all signatures were logically equivalent to the source code.

Signature quality: Note, all regexes that have a corresponding traffic trace generates valid a match with the traffic trace. Here, we attempt to quantify how many constant keywords are present in our regex signature of request/response bod-

[5]Currently, PUMA is the most advanced UI automation tools for Android apps that are publicly available.

	Request Body/ Query String	Response Body (R_k, R_v, R_n)
Open-source apps	47/52/1%	7/48/45%
closed-source apps	48/31/21%	16/35/49%

Table 2: Matched byte count % on actual traffic

ies. For this, we compare the number of constant keywords found in our signature with that in the packet trace obtained by manual fuzzing. We count the keys in key-value pairs of query string, JSON bodies, the tags, and attributes in XML bodies that appear in the traffic trace. For open source apps, we also obtain the ground truth by counting the number of constant keywords appeared in the source code.

Figure 7 shows the number of constant keywords identified. In the request bodies of open-source apps, a total of 145 keywords are identified through manual source code analysis and manual UI-fuzzing. Extractocol identifies all but one. In RRD, a JSON key-value pair string is generated from an user input and stored in a heap object. At a later time, another event triggers an HTTP request. Because we did not use the heuristics for open source apps in §3.4, Extractocol cannot identify implicit dependencies by tracking across multiple asynchronous events. However, Extractocol successfully covers this case when the option enabled. For closed-source apps, the traffic traces from manual fuzzing only contained 3,507 keywords while Extractocol identified 7,793.

We verify the response body signatures in the same way. For closed-source apps, Extractocol identifies 14K JSON keys, while traffic contains 13K. But the gap has decreased from the signature count in Figure 6. This is because response bodies contain JSON keys that are often dynamically generated. For example, a JSON object can contain a list of products as JSON arrays with key being the product ID and value being the product name. For open source apps, responses from the packet traces contain 616 keywords. Extractocol and manual source code analysis identify 60% of them. Upon inspecting the source code, we find that some apps do not inspect all keywords contained in JSON (§3). Despite this, for open source apps with ground truth, all response body signatures were equivalent to the source code.

To further quantify the signature quality, we count the number of bytes that matches with the constant and wildcard of our regex in request body/query string a nd response bodies. R_k and R_v respectively denote the fraction of matched byte count on the constant keywords and the corresponding value parts of our signature (often wildcards). R_n is the fraction of byte count that whose key and values are both wild cards. Table 2 shows the overall fraction of byte count. For open-source apps, 70% bytes of request URIs match with the constant part of our signature (indicated by R_k), and 65% bytes match in closed-source apps. For query strings and request bodies, 100% (R_k + R_v) of all bytes for open-source apps and 91% for closed-source apps match with key-value pairs identified by Extractocol. The rest of 9% (R_n) is covered by the wildcard part of our regex signature. For response body in open-source and closed-source apps, 55% and 51% of bytes match with them respectively, and the rest is covered by wildcards in our regex.

5.2 Characterization of App Behavior

To demonstrate Extractocol's effectiveness in characterizing app's protocol behavior, we present analysis result in greater detail for TED and radio reddit [20] that perform media streaming.

TED is one of the "Best Apps of 2014" on Google Play. We choose this app because it provides rich UI, contains dynamically generated pages including advertisements, and uses third-party libraries, such as the Facebook API.

Extractocol identifies 18 HTTP(S) request and 10 response body signatures (8 responses have no bodies that are processed by the app). Table 4 summarizes notable transactions and their relationships with dependency graph identified by Extractocol. Requests for transaction #1, #3 and #6 use constant string with an *api-key*, which is stored in `android.content.res.Resourcesclass`. It processes their JSON responses and then inserts (or updates if exists) them into a database (`android.database.sqlite.SQLiteDatabase`).

Transaction #1 and #6 show their update operations, and these updated values are later processed in transaction #7 and #8 for their requests, respectively (*Thumbnail URI* and *Audio/video URI* columns of the database). Meanwhile, after parsing JSON responses from transaction #3, transaction #4 retrieves an advertisement URI string, which is then processed for transaction #4's request without database updates. In the same manner, transaction #5 obtains advertisement video URLs for requesting advertisement video streams. Since Extractocol provides Android resource and database semantics, if a data dependency exists between transactions and semantic models such as resources and databases, we are able to identify these transactional relationships. To verify the result with real network traces, we execute TED using PUMA [54], a state-of-the-art automated UI testing tool. Note our system and PUMA have different goals. PUMA merely iterates through all "clickable" elements in the UI to generate all protocol messages, but does *not* identify the relationship between protocol messages unlike ours. PUMA generated a total of 158 HTTP(S) requests. PUMA took 10.3 minutes and Extractocol took 132.5 minutes to complete.

For easy comparison, first we manually group the request URIs into unique patterns. We identify 65 unique URI patterns; URIs in each pattern shared either prefix or postfix. Next, since multiple dynamic traces can map to a single signature, we manually classify each trace into static or dynamic. If all of the URI string are found in a prior response, we classify the request as dynamic. If any part of the URI (delimited by slash) matches with strings in the decompiled APK, we classify it as static. For dynamic requests, we identify their origin (i.e., sources of their dynamic URI).

Using the dataset, we test whether the request URIs and responses obtained from PUMA matches with our signatures, and our transactional relationships match network traffics generated by PUMA. All signatures match with their corresponding traffic traces, and all transactional relationships

```
HTTP Response URI
GET http://www.radioreddit.com/api/hiphop/status.json

HTTP Response Body
[{ "all_listeners":"99999", "listeners":"13586", "online":"TRUE",
  "playlist":"hiphop",
  "relay":"http://cdn.audiopump.co/radioreddit/hiphop_mp3_128k",
  "songs":{ "song":[{
    "album": "", "artist": "stirus", "download_url": …(omitted),
    "genre": "Hip-Hop", "id": "837", "preview_url": …(omitted),
    "reddit_title": "stirus(\/u\/sonus) - Surviving Minds",
    "reddit_url": …(omitted), "redditor": "sonus",
    "score": "6", "title": "Surviving Minds" }]}
}]
```

Figure 8: Traffic trace for RRD transaction #2

also match with their corresponding network traffic traces. Surprisingly, Extractocol identified more than 4 transactions than PUMA did. For example, PUMA did not identify a request triggered in response to content updates, triggered by the server. Such externally triggered events are hard to reproduce using dynamic analysis.

Radio reddit (RR) is an online music streaming client that allows users to choose radio stations and vote on or save songs using their reddit accounts. Table 3 shows the result for RR with six transactions; five of them use HTTP, while login request uses HTTPS. It also shows a dependency graph identified by Extractocol. Extractocol identifies that login request (#3) includes three fields, and its response is a JSON object including "modhash" and "cookie" as keys. The two fields in login response (#3) are used in #4's and #5's requests. They use the "modhash" value in the "uh" field, and add the "cookie" value to their request headers. We verify that the identified information accurately corresponds to reddit's API documentation Extractocol identified the exact API subset that is actually used by the app. It even identified a hidden API *not* discussed in the official document [66].

Figure 8 shows the actual traffic trace of transaction #2 with constant keywords highlighted from our regex representing the JSON's tree structure. The URI signature contains all strings in the URI except "api/hiphop" which is obtained via user input. The response regex signature contains 16 keywords out of 18 in the response; two keywords ("album" and "score") are not processed by the app. The app then passes the station's "relay" URI (Figure 8) to Android's `MediaPlayer`, which generates transaction #6.

5.3 Reverse-Engineering

Manual reverse-engineering attempts use packet traces to analyze popular REST APIs [22, 25, 42, 43, 74, 83]. We use Kayak to demonstrate Extractocol's practical usefulness in reverse-engineering the API syntax. Kayak API is a private REST API used by Kayak.com, a popular fare comparison web site. Its API used to be public, but its service was recently discontinued [22], and a repurposed private API was introduced. We compare our analysis results with a manual analysis result in [22], which lists three APIs related to flight fare comparison.

We only scope the analysis to `com.kayak` classes excluding the external libraries to focus on its API. Extractocol identifies a total of 46 HTTP(S) transactions, including 39 GET and 7 POST requests, 6 JSON responses, and other responses (e.g., text and images). It identifies all three APIs from prior manual analysis [22]. Additionally, Extractocol reveals 14 times more APIs in just 31 minutes. It also identifies the use of app-specific HTTP header, "User-Agent : kayakandroidphone/8.1". Table 5 shows a summary of requests grouped into eight categories by their URI prefix. Table 6 shows three selected APIs from the results, demonstrating that Extractocol identifies URIs and query strings used by the app.

To verify the results, we implement a simple Python script code (73 LOC) that generates HTTPS requests for flight fare comparison (Table 6) based on our signatures. It first sends a '/k/authajax' request to start a new session using the app-specific 'User-Agent' field. It then sends '/flight/start' and '/flight/poll' requests. We verify that it successfully retrieves flight fare information. We find that the APIs are slightly different from [22] due to the difference in platform (e.g., 'action=registerandroid' for '/k/authajax'). We also find that the 'User-Agent' header that we identified is important because Kayak performs access control using the header to prevent unauthorized access from other platforms.

Summary: Extractocol presents the first automatic and comprehensive protocol analysis for Android apps. It provides higher coverage than dynamic fuzzing and even manual fuzzing. We demonstrate the combination of signature extraction, pairing, and dependency analysis produces a rich characterization. We believe the rich information it provides is valuable to a number of potential applications, including understanding apps, app testing, and reverse-engineering. In fact, Extractocol has been already used to build application acceleration proxies for mobile apps [35].

Discussions: Our evaluation provides a comprehensive analysis of a small number of applications, rather than a large-scale analysis. Although Extractocol provides automatic analysis, large-scale evaluation is still challenging because verifying the result requires manual analysis of signatures and their dependencies takes a long time since most apps are obfuscated. Note, even generating all protocol messages requires manual intervention due to limitations in UI-automation (§5.1). Second, Extractocol does not support binary protocols and its support for implicit callbacks is not complete (§3.4). Finally, to handle obfuscated 3rd party libraries, Extractocol preprocesses the code to generate a map between the obfuscated identifier using heuristics (§3.4). However, if the mapping is inaccurate, Extractocol may not be able to identify the correct semantics and may generate wild card ("*") signatures instead of accurate ones. We expect that the limitations can be overcome by incorporating it with techniques [9, 12, 33, 62] discussed in §3.4.

6 Related Work

This work builds on a large body of work that performs automated program analysis for Android applications. An earlier poster version of this work can be found in [36].

Android program analysis: Prior work on Android program analysis focuses on discovering privacy-sensitive information leakage [27, 34, 46, 51, 53, 57, 63, 69, 81, 86, 87], identifying

#	Request URI	Request Body	Dependency graph
1	GET (http://www.reddit.com/api/info.json?)	-	
2	GET (http://www.radioreddit.com/)(status.json)	-	
3	POST (https://ssl.reddit.com/api/login)	(user=).*(&passwd=) (&api_type=json)	
4	POST (http://www.reddit.com/api/)(unsave | save)	(id=).*(&uh=)(.*)	
5	POST (http://www.reddit.com/api/vote)	(id=).*(&dir=).*(&uh=)(.*)	
6	GET (.*)	(.*)	

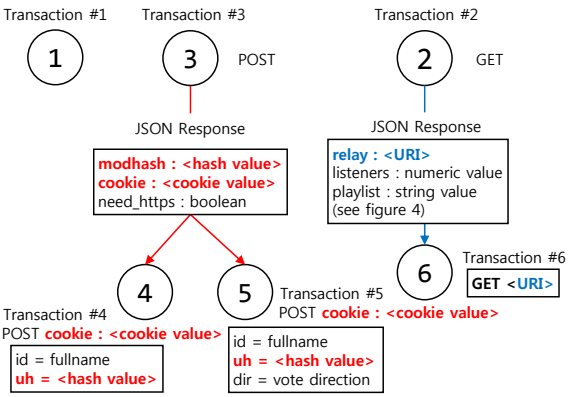

Table 3: Reconstructed HTTP transactions and their dependency graph for Radio reddit

#	Request (Static/Dynamically-derived URI)	Response	Dependency graph
1	Speaker's info (S)	JSON name/description inserted to DB	
2	Facebook sharing (S)	String	
3	Advertisement query (S)	JSON Ad query URI	
4	GET (.*): Ad query URI from #3 (D)	XML Ad resource URIs	
5	GET (.*): Ad video URI from #4 (D)	Binary	
6	Talk info (S)	JSON thumbnail/video URIs inserted to DB	
7	GET (.*): Thumbnail URI from DB (D)	Binary	
8	GET (.*): Audio/video URI from DB (D)	Binary	

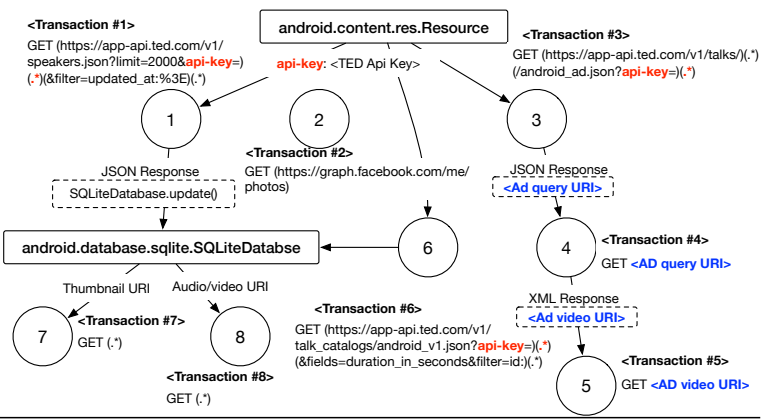

Table 4: Selected HTTP transactions and its dependency graph for TED

application misbehaviors [45, 55, 76], or providing privilege separation for apps [50].

Taint analysis tracks information flows to reveal unintended information leakage. TaintDroid [46] performs real-time dynamic taint analysis to detect privacy-sensitive information leakage on Android. While it is more accurate (i.e., low false positives) than static program analysis, achieving high coverage for ahead-of-time analysis is an important challenge. To overcome this, many studies employ static analysis [27, 51, 86]. These studies commonly reconstruct the inter-procedural control flow graph (ICFG) by modeling Android app's life-cycle. By analyzing the ICFG and data dependencies, they identify whether a path exists from a source of sensitive information to a taint sink (usually a network interface). In this work, we leverage FlowDroid [27] to reconstruct the ICGF and use data dependencies in three different ways for protocol analysis. Finally, CryptoLint [45] detects the misuse of cryptographic libraries using static program slicing. SMV-Hunter [76] identifies Android applications that fail to properly validate SSL certificates. Fratantonio et al. [50] use symbolic execution to enable finer-grained access controls in Android applications.

Finger-printing Android app traffic: FLOWR [84, 85] tries to classify mobile app traffic by extracting key-value pairs from HTTP sessions. NetworkProfiler [41] uses UI-based fuzzing on Android apps to build a comprehensive network trace. The main focus is to generate network traffic and extract unique finger-prints, rather than protocol analysis.

A large body of work addresses protocol analysis for conventional, non-mobile applications. We describe them below.

Protocol analysis using network traces: Many studies [26, 29, 38, 39, 49, 58, 59, 78, 80, 84] use traffic traces as input to derive application protocol information, such as protocol syntax and state machine. Discoverer [38] infers message format to derive application protocol syntax, and ASAP [58] uses machine learning to extract typical communication signatures and their semantics. RolePlayer [39] and ScripGen [59] support automatic protocol replay by reconstructing one side of an application session in different contexts by adjusting protocol fields, such as ports, cookies, and sequence numbers. However, this approach inherently relies on pattern inference which is less accurate [30] than program semantic analysis and cannot handle encrypted messages. Also, a common limitation of the approach is that obtaining a sufficient size

Cat.	Method	URI Prefix	# APIs	Example Sub URI	Response
Travel Planner	GET	https://www.kayak.com/trips/v2	11	/edit/trip/	-
Authentication	POST	https://www.kayak.com/k/authajax	4	-	-
Facebook Auth	POST	https://www.kayak.com/k/run/fbauth	2	/login	-
Flight	GET	https://www.kayak.com/api/search/V8/	6	/flight/start	JSON
Hotel	GET	https://www.kayak.com/api/search/V8/	2	/hotel/detail	JSON
Car	GET	https://www.kayak.com/api/search/V8/	1	/car/poll	JSON
Mobile Specific	GET	https://www.kayak.com/h/mobileapis	12	/currency/allRates	JSON
Advertising	GET	https://www.kayak.com/s/mobileads	1	-	JSON
Etc.	POST	https://www.kayak.com/k	4	/cookie	-

Table 5: A summary of Kayak API analysis results

Sub URIs	Query String
/k/authajax	action=registerandroid&uuid=.*&hash=.*&model=.* &platform=android&os=.*&locale=.*&tz=.*
/api/search /V8/flight/start	cabin=.*&travelers=.*&origin=.*&nearbyO.*=.* &destination.*=.* &nearbyD.*=.*&depart_date.*=.* &depart_time.*=.*&depart_date_flex.*=.* &_sid_=.*
/api/search /V8/flight/poll	searchid=.*&nc=.*&c=.*&s=.*&d=up&currency=.* &includeopaques=true &includeSplit=false

Table 6: Selected Request Signatures for Kayak

of input that exhaustively contains protocol messages is hard especially due to highly skewed message popularity [38].

Protocol analysis using program analysis: Many studies identify application network protocol using program analysis [30–32, 37, 40, 56, 61, 82]. The majority of program analysis techniques [30–32, 37, 40, 61, 82] use dynamic analysis as primary means to extract protocol information with different goals. Dispatcher [30] reverse engineers a botnet's command-and-control (C&C) protocol to actively rewrite C&C messages. Replayer [68] and Rosetta [31] focus on replaying application dialogs, while others [32, 37, 40, 61] aim for identifying protocol fields within a message. Prospex [37] extracts full protocol specification including the state machine. These approaches take protocol messages as input to generate execution traces, use heuristics to decide the field boundary within the message by inspecting the execution traces, and generalize the observation using various inference techniques to infer a generic message format and protocol state. The limitations of dynamic approaches are that 1) they only identify limited information, such as the field or delimiters (i.e., the goal is output fields similar to that used in Wireshark); 2) can only analyze one-side of communication (i.e., can only infer the format of received message) [6]; 3) require the input messages to exhaustively cover all message types; 4) require multiple messages instances per type for inferring general protocol format [37, 40, 61, 82]; and 5) rely on pattern inference for reconstructing client protocol state machine. In contrast, Extractocol provides richer behavioral information. To our best knowledge, it is the first to provide such in-depth behavior analysis. Finally, StateAlyzr [56] applies static analysis techniques to automatically identify states that need to be transferred when replicating middlebox states across virtualized network function instances. It takes middlebox application code to identify all state that must be migrated/cloned to ensure consistent middlebox output in the face of redistribution/cloning. It also uses program-slicing technique to derive the dependencies between state objects and packet processing logic in the source code of middlebox applications. In terms of techniques used, the two main differences between StateAlyzr and Extractocol are: (1) our framework operates on binary whereas StateAlyzr takes source code as input, and (2) Extractocol not only extracts slices but it extracts their semantics to reconstruct protocol messages.

[6]Analyzing both sides requires combining server and client analysis. However, existing work falls short in analyzing two-way communication. For Android apps, the server binary is often unavailable.

7 Conclusions

This work presents the first system for automatic analysis of HTTP(S)-based application protocol behaviors for Android apps. Extractocol uses the application binary as input to reconstruct application-specific HTTP-based interactions using static program analysis. It combines network-aware static taint analysis and semantic analysis to provide a comprehensive characterization of application protocol behaviors. Extractocol provides a rich analysis of app behaviors, including message signatures, request-response pairs, and inter-transactional dependencies. We expect that this provides valuable information for many applications [35]. Our in-depth evaluation on open-source and closed-source apps demonstrate that 1) it provides high coverage and accuracy in identifying protocol messages; 2) it provides rich characterization of app behavior; 3) it is capable of reverse-engineering REST APIs; and 4) it can automatically analyze many applications. Finally, we believe Extractocol and its approach can serve as a basis for generic protocol analysis (other than HTTP) for Android applications.

Acknowledgement

We would like to thank our shepherd Alessandro Finamore and anonymous reviewers for their feedback and suggestions. This work is in part supported by IITP funded by the Korea government (MSIP) under B0126-16-1078 and under IITP-2016-R0992-16-1006.

8 References

[1] Android library statistics - appbrain. http://www.appbrain.com/stats/libraries. Accessed Sep. 2015.

[2] Bitblaze: Binary analysis for computer security. http://bitblaze.cs.berkeley.edu/. Accessed Jun, 2016.

[3] Connecting to the network. https://developer.android.com/training/basics/network-ops/connecting.html. Accessed Jun. 2016.

[4] Dynamic Site Acceleration explained. https://aiscaler.com/dsa-explained#prefetching. Accessed Sep. 2015.

[5] Dynamic site accelerator. https://www.akamai.com/kr/ko/multimedia/documents/product-brief/dynamic-site-accelerator-product-brief.pdf. Accessed Oct. 2016.

[6] Eliminate Blind Spots in SSL Encrypted Traffic. https://www.venafi.com/assets/pdf/sb/SSL_Visibility_Solution_Brief.pdf.

[7] Extending prefetching to json and other objects. https://community.akamai.com/community/web-performance/blog/2015/02/19/extending-prefetching-to-json. Accessed Sep. 2015.

[8] Free and open source android app repository. https://f-droid.org.

[9] Generic android deobfuscator. https://github.com/CalebFenton/simplify. Accessed Jun. 2016.

[10] How antivirus software works: Virus detection techniques. http://searchsecurity.techtarget.com/tip/How-antivirus-software-works-Virus-detection-techniques. Accessed Oct. 2016.

[11] How dynamic site acceleration works: What at&t and akamai offer. http://blog.streamingmedia.com/2010/10/how-dynamic-site-acceleration-works-what-akamai-and-cotendo-offer.html. Accessed Sep. 2015.

[12] A pattern based dalvik deobfuscator which uses limited execution to improve semantic analysis. https://github.com/CalebFenton/dex-oracle. Accessed Jun. 2016.

[13] SSL Decryption. https://www.gigamon.com/products/technology/ssl-decryption. Accessed Oct. 2016.

[14] SSL Encrypted Traffic Visibility and Management. https://www.bluecoat.com/products/ssl-encrypted-traffic-visibility-and-management. Accessed Oct. 2016.

[15] Top applications on google play. https://www.appannie.com/apps/google-play/top-chart/united-states/application/. Accessed Sep. 2015.

[16] An accuweather cloudlet answers a hail of data requests. https://www.yumpu.com/en/document/view/23074646/an-accuweather-cloudlet-answers-a-hail-of-data-requests, 2011.

[17] Android async http clients: Volley vs retrofit. http://instructure.github.io/blog/2013/12/09/volley-vs-retrofit/, 2013.

[18] Number of available android applications. http://www.appbrain.com/stats/number-of-android-apps, 2014.

[19] Proxydroid - android apps on google play. https://play.google.com/store/apps/details?id=org.proxydroid&hl=en, Dec. 2014.

[20] radio reddit v0.7. https://f-droid.org/repository/browse/?fdfilter=radio+reddit&fdid=com.radioreddit.android, 2014. An open-source radio reddit app.

[21] Report: Apple iphone drives half of all mobile internet traffic. http://marketingland.com/report-apple-iphone-drives-half-mobile-internet-traffic-111129, 2014.

[22] Reverse-engineering the kayak app with mitmproxy. http://www.shubhro.com/2014/12/18/reverse-engineering-kayak-mitmproxy/, Dec. 2014.

[23] mitmproxy. https://mitmproxy.org/, 2015.

[24] Proguard, android developers. http://developer.android.com/tools/help/proguard.html, 2015.

[25] M. Amps. Reverse engineering shopify private apis. http://ma.rtin.so/reverse-engineering-shopify-private-apis, 2013.

[26] J. Antunes and N. Neves. Automatically complementing protocol specifications from network traces. In *European Workshop on Dependable Computing*, 2011.

[27] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN PLDI*, 2014.

[28] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, 2012.

[29] M. A. Beddoe. Network protocol analysis using bioinformatics algorithms. http://www.4tphi.net/~awalters/PI/PI.html, 2004.

[30] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *ACM CCS*, 2009.

[31] J. Caballero and D. Song. Rosetta: Extracting protocol semantics using binary analysis with applications to protocol replay and nat rewriting, 2007. Technical Report, UC Berkeley.

[32] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *ACM CCS*, 2007.

[33] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[34] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *ACM MobiSys*, 2011.

[35] B. Choi, J. Kim, and D. Han. Application-specific Acceleration Framework for Mobile Applications. In *ACM SIGCOMM (poster session)*, 2016.

[36] H. Choi, J. Kim, H. Hong, Y. Kim, J. Lee, and D. Han. Extractocol: Automatic extraction of application-level protocol behaviors for android applications. In *ACM SIGCOMM (poster)*, pages 593–594. ACM, 2015.

[37] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *IEEE Symposium Security and Privacy*, 2009.

[38] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security*, 2007.

[39] W. Cui, V. Paxson, N. C. Weaver, and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *NDSS*, 2006.

[40] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *ACM CCS*, 2008.

[41] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. Networkprofiler: Towards automatic fingerprinting of android apps. In *IEEE INFOCOM*, 2013.

[42] N. Derkach. Reverse engineering the private api: Hacking your couch. http://www.toptal.com/back-end/reverse-engineering-the-private-api-hacking-your-couch, 2014.

[43] T. Dorr. Unofficial documentation of the Tesla Model S JSON API. http://docs.timdorr.apiary.io/, 2013.

[44] P. Edholm. Reverse-engineering the kayak app with mitmproxy. http://www.nojitter.com/post/240153039/hp-and-microsoft-demo-openflowlync-applicationsoptimized-network. Apr. 2013.

[45] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *ACM CCS*, 2013.

[46] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An

information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX OSDI*, 2010.

[47] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security*, 2011.

[48] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A First Look at Traffic on Smartphones. In *ACM Internet Measurement Conference*, 2010.

[49] J. François, H. Abdelnur, R. State, and O. Festor. Automated behavioral fingerprinting. In *International Symposium on Recent Advances in Intrusion Detection*, 2009.

[50] Y. Fratantonio, A. Bianchi, W. Robertson, M. Egele, C. Kruegel, E. Kirda, and G. Vigna. On the Security and Engineering Implications of Finer-Grained Access Controls for Android Developers and Users. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2015.

[51] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *International conference on Trust and Trustworthy Computing*, 2012.

[52] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-Flow Analysis of Android Applications in DroidSafe. In *NDSS*, 2015.

[53] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *NDSS*, 2012.

[54] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *ACM Mobisys*, 2014.

[55] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. AsDroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *International Conference on Software Engineering*, 2014.

[56] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the Way for NFV: Simplifying Middlebox Modifications using StateAlyzr. In *USENIX NSDI*, 2016.

[57] J. Kim, Y. Yoon, K. Yi, J. Shin, and S Center. ScanDal: Static analyzer for detecting privacy leaks in android applications. In *Mobile Security Technologies*, 2012.

[58] T. Krueger, N. Krämer, and K. Rieck. Asap: Automatic semantics-aware analysis of network payloads. In *International Workshop on Privacy and Security Issues in Data Mining and Machine Learning*, 2011.

[59] C. Leita, K. Mermoud, and M. Dacier. Scriptgen: An automated script generation tool for honeyd. In *Annual Computer Security Applications Conference*, 2005.

[60] O. Lhoták and L. Hendren. Scaling Java Points-to Analysis Using SPARK. In *International Conference on Compiler Construction*, 2003.

[61] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *NDSS*, 2008.

[62] B. Liu, B. Liu, H. Jin, and R. Govindan. Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps. In *ACM MobiSys*, 2015.

[63] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *ACM CCS*, 2012.

[64] G. Maier, F. Schneider, and A. Feldmann. A First Look at Mobile Hand-held Device Traffic. In *International Conference on Passive and Active Measurement*, 2010.

[65] mttkay. https://mttkay.github.io/blog/2013/08/25/functional-reactive-programming-on-android-with-rxjava/, 2013.

[66] J. Musser. Reddit's secret api. http://www.programmableweb.com/news/reddits-secret-api/2008/11/25, 2008.

[67] D. naylor, K. Schomp, M. Varvello, I. Leontiadis, D. L. Jeremy Balckburn, P. R. R. Konstantina Papaginnaki, and P. Steenkiste. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *ACM SIGCOMM*, 2015.

[68] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic protocol replay by binary analysis. In *ACM CCS*, 2006.

[69] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in Android with Epicc: an essential step towards holistic security analysis. In *USENIX Security*, 2013.

[70] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein. Analyzing protocol implementations for interoperability. In *USENIX NSDI*, 2015.

[71] Z. A. Qazi, J. Lee, T. Jin, G. Bellala, M. Arndt, and G. Noubir. Application-awareness in SDN. In *ACM SIGCOMM*, 2013.

[72] C. Qian, X. Luo, Y. Shao, and A. T. S. Chan. On Tracking Information Flows Through JNI in Android Applications. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[73] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995.

[74] S. Saffron. How to reverse engineer the discourse api. https://meta.discourse.org/t/how-to-reverse-engineer-the-discourse-api/20576, 2014.

[75] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *USENIX OSDI*, 2004.

[76] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *NDSS*, 2014.

[77] A. Tongaonkar, R. Keralapura, and A. Nucci. Challenges in network application identification. In *LEET*, 2012.

[78] A. Trifilò, S. Burschka, and E. Biersack. Traffic to protocol reverse engineering. In *IEEE Symposium on Computational Intelligence for Security and Defense Applications*, 2009.

[79] N. Viennot, E. Garcia, and J. Nieh. A Measurement Study of Google Play. In *ACM SIGMETRICS*, 2014.

[80] Y. Wang, Z. Zhang, D. D. Yao, B. Qu, and L. Guo. Inferring protocol state machine from network traces: A probabilistic approach. In *International Conference on Applied Cryptography and Network Security*, 2011.

[81] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *ACM CCS*, 2014.

[82] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *NDSS*, 2008.

[83] J. Wright. Reverse Engineering the We Heart It API. http://jordan-wright.github.io/blog/2014/10/12/reverse-engineering-the-we-heart-it-api/, 2014.

[84] Q. Xu, T. Andrews, Y. Liao, S. Miskovic, Z. M. Mao, M. Baldi, and A. Nucci. Flowr: A self-learning system for classifying mobile application traffic. In *ACM SIGMETRICS*, 2014.

[85] Q. Xu, Y. Liao, S. Miskovic, Z. M. Mao, M. Baldi, A. Nucci, and T. Andrews. Automatic generation of mobile app signatures from traffic observations. In *IEEE INFOCOM*, 2015.

[86] Z. Yang and M. Yang. LeakMiner: Detect information leakage on Android with static taint analysis. In *World Congress on Software Engineering*, 2012.

[87] M. Zhang and H. Yin. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *NDSS*, 2014.

[88] T. Zinner, M. Jarschel, A. Blenk, F. Wamser, and W. Kellerer. Dynamic application-aware resource management using Software-Defined Networking: Implementation prospects and challenges. In *IEEE/IFIP Network Operations and Management Symposium*, 2014.