석사 학위논문
Master’s Thesis

안드로이드 애플리케이션의 네트워크 행동패턴 분석 자동화 연구
Enabling Automatic Protocol Behavior Analysis for Android Applications

김 정 민 (金政敏  Kim, Jeong-Min)
정보보호대학원
Graduate School of Information and Security

KAIST

2016
Enabling Automatic Protocol Behavior Analysis for Android Applications
Enabling Automatic Protocol Behavior Analysis for Android Applications

Advisor : Professor Han, Dong Su

by
Kim, Jeong-Min
Graduate School of Information and Security
KAIST

A thesis submitted to the faculty of KAIST in partial fulfillment of the requirements for the degree of Master of Science in the Graduate School of Information and Security. The study was conducted in accordance with Code of Research Ethics.

2015. 12. 1.

Approved by
Professor Han, Dong Su
[Advisor]

---

1Declaration of Ethical Conduct in Research: I, as a graduate student of KAIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.
안드로이드 애플리케이션의 네트워크 행동패턴 분석 자동화 연구

김 정 민

위 논문은 한국과학기술원 석사학위논문으로 학위논문심사위원회에서 심사 통과하였음.

2015년 12월 1일

심사위원장  한 동 수  (인)

심사위원  김 용 대  (인)

심사위원  신 승 원  (인)
ABSTRACT

Android application is an important class on today’s Internet. While understanding app-specific behavior is important for network operation and management, 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. Extractocol only uses Android application binary as input and accurately reconstructs HTTP transactions (request-response pairs) and identifies their message format and relationships using binary analysis. Our evaluation and in-depth case studies on commercial and open-source apps demonstrate that Extractocol provides high coverage and accurately characterizes network-related application behaviors.
# Contents

Abstract ................................................................. i  
Contents ............................................................... ii  
List of Tables ......................................................... iii  
List of Figures ......................................................... iv  

Chapter 1. Introduction ............................................. 1  

Chapter 2. Motivation and Approach ............................ 3  
  2.1 Motivation ......................................................... 3  
  2.2 Goal ................................................................. 4  
  2.3 Non-goal ............................................................ 4  
  2.4 Approach .......................................................... 4  

Chapter 3. Extractocol Design .................................... 5  
  3.1 Network-aware Program Slicing .............................. 5  
  3.2 Signature Extraction ........................................... 7  
  3.3 Message Dependency Analysis ............................... 10  
  3.4 Handling Asynchronous Events .............................. 12  

Chapter 4. Implementation ....................................... 13  

Chapter 5. Evaluation .............................................. 14  
  5.1 Validation of Protocol Analysis .............................. 14  
  5.2 Characterization of App Behavior ............................ 18  
  5.3 Reverse-Engineering ............................................ 19  

Chapter 6. Related work ......................................... 21  

Chapter 7. Conclusions ............................................ 23  

References ............................................................. 24  

Summary (in Korean) ................................................ 30
List of Tables

5.1 Open-source and commercial App (gray box) test case summary .................................. 14
5.2 Coverage: # of request/response produced by Extractocol and % of identified signatures
    compared to the ground truth ...................................................................................... 15
5.3 The Summary URLs of AOL ....................................................................................... 16
5.4 Matched byte count % on actual traffic ...................................................................... 17
5.5 Selected HTTP transactions and its dependency graph for TED ................................. 18
5.6 Reconstructed HTTP transactions and their dependency graph for Radio reddit ...... 19
5.7 (Radio) reddit [7, 8, 53] API document ..................................................................... 19
5.8 Traffic trace for RRD transaction #2 .......................................................................... 19
5.9 A summary of Kayak API analysis results ................................................................... 20
5.10 Selected Request Signatures for Kayak ..................................................................... 20
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Application acceleration example</td>
<td>3</td>
</tr>
<tr>
<td>3.1</td>
<td>Design overview of Extractocol</td>
<td>5</td>
</tr>
<tr>
<td>3.2</td>
<td>Reddit Client (Diode)'s Request &amp; Response slice example</td>
<td>6</td>
</tr>
<tr>
<td>3.3</td>
<td>A simple language to represent signature patterns in Extractocol</td>
<td>8</td>
</tr>
<tr>
<td>3.4</td>
<td>A subgraph of ICFG of Diode on a demarcation point. The request/response slices are highlighted</td>
<td>10</td>
</tr>
<tr>
<td>3.5</td>
<td>Signature outputs and control flow graph in basic block from the request slices of Diode</td>
<td>10</td>
</tr>
<tr>
<td>3.6</td>
<td>Extractocol locates disjoint code segments, between which only a single path exists</td>
<td>11</td>
</tr>
<tr>
<td>5.1</td>
<td>Open source apps # signatures (left) # keywords (right)</td>
<td>17</td>
</tr>
<tr>
<td>5.2</td>
<td>Commercial apps # signatures (left) # keywords (right)</td>
<td>17</td>
</tr>
</tbody>
</table>
Chapter 1. Introduction

Android app is an important class of today’s Internet applications that generate roughly 40-50% of mobile Web and app traffic [13]. More than 1.4 million Android apps are offered through Google’s open market [10], and tens of thousands of new apps are added every month.

Understanding the application behavior within the network is often crucial in providing value-added services, such as application acceleration [3] and dynamic caching [6]. However, very little information is known about Android application protocol behaviors because they predominantly use proprietary protocols on top of HTTP [32, 39, 52, 62]. 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-layer messages for each individual application.

However, existing systems fall short in delivering comprehensive analysis of fine-grained application layer interaction. Existing works [32, 69] that try to analyze Android application protocol merely focus on automatic generation of application traffic and extracting unique fingerprints that can be used to identify the application traffic. Manual analysis (e.g., reversing) is not an option because it takes tremendous amount of human effort. In Android, the popular use of and default support for code obfuscation tools, such as ProGuard, make this even more difficult [16, 38].

Prior work can be categorized into two approaches: traffic trace analysis and dynamic binary analysis. Traffic trace analysis takes large amounts of network traffic that contains many instances of message exchanges as input to infer protocol syntax [18, 21, 29, 46, 63, 64, 69]. One major limitation is that such an exhaustive traffic trace is often very difficult to obtain, especially due to highly skewed message popularity [29] and the sheer number of constantly evolving apps. Dynamic analysis uses execution traces generated by executing apps with appropriate network traffic [19, 24, 28]. For high coverage, it requires multiple execution traces generated using multiple message types, because a single execution trace only covers on path out of all possible paths [19, 24]. Even for a single message type, it requires multiple execution instances to generalize the observation [28, 66]. Both of these approaches are often used with UI-based fuzzing that automatically generates messages [32, 69]. However, even UI-based fuzzing [32, 43] does not generate all messages, as we demonstrate in

This paper presents the first comprehensive protocol analysis framework that automatically extracts protocol behaviors, formats, and message signatures by leveraging binary analysis. Using binary analysis for Android application protocol analysis, however, requires solving two unique challenges that have not been addressed previously. First, shile 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 [24, 54] or even the source code [56]. Second, it must provide high coverage and accurately infer the message format and relationship between messages. This is important because protocol messages often have dependencies—e.g., an authentication token to be used for subsequent requests may be embedded in a prior login response, or a URI (e.g., image ID) may be embedded in a prior response.

Our approach is to use the application binary as input and track the code that generates or processes
network messages. The key insight is to obtain dependency relationships of objects that flow in from and flow out to the network using static binary analysis. In particular, Extractocol which is an earlier version of this work appeared in [27] 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 regular expression 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. It 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.

- **New techniques for inferring message relationships and signature generation:** 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 commercial apps demonstrates that Extractocol provides a rich and comprehensive characterization.
Chapter 2. Motivation and Approach

2.1 Motivation

Understanding the behavior of network applications has significant implications in networking. Apart from its intrinsic value, it enables the opportunity for the network to provide value-added services, often in conjunction with other networking technologies.

We provide two examples of how our protocol analysis can be applied to enable new services. First, it can enable app-specific acceleration in mobile networks. 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 [3, 4, 6]. Figure 2.1 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 [6, 9]. Today, the development of dynamic caching proxies is done manually on a per-app basis [9] because it requires the knowledge of application semantics (e.g., which request parameter is dynamically generated) to determine which content is cacheable [6]. We believe this can be automated with automatic protocol analysis and Extractocol is the first step towards this.

Second, it can enable application-aware treatment. 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. We believe that the understanding of application behavior combined with advances in software-defined networking [35, 57, 72] can enable new dimensions of application-aware networking.

Figure 2.1: Application acceleration example

Second, it can enable application-aware treatment. 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. We believe that the understanding of application behavior combined with advances in software-defined networking [35, 57, 72] can enable new dimensions of application-aware networking.
2.2 Goal

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 between messages; and 3) enables analysis even when apps use HTTP. Our goal is to provide a comprehensive analysis of each individual application, rather than a large-scale analysis.

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.

2.3 Non-goal

By definition, REST is stateless—the server does not keep session state. Thus, we do not model the server state machine. Note, while each request is self-descriptive and each transaction is independent, transactions may have dependencies—e.g., login providing an access token to be used by subsequent requests. Extractocol does track such dependencies by inferring the data flow between transactions.

Our system only analyzes Dalvik VM bytecode. This includes third-party code, written using Android or Java APIs. However, it does not analyze native binary. Existing frameworks that specialize on native code, such as the BitBlaze project [2], can be applied to handle the use of Java Native Interface (JNI) on Android. However, we do not address this in the paper. Note a recent study on 7.4K Android apps reveals that a relatively small fraction (7%) of them uses native binary [41].

Finally, our focus in this paper is an in-depth analysis of protocol behaviors for individual applications, rather than a large-scale, overall characterization. We believe the nature of protocol analysis fits better with the scope, and an in-depth automatic analysis of protocol behavior itself is a worth-while goal that is valuable in practice.

2.4 Approach

Our key insight in protocol analysis is to track all objects that eventually go out to or flow in from the network. 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. For tracking protocol-related data flow, we extend FlowDroid [19] that provides flow-, context-sensitive, and inter-procedural data flow analysis on Android apps.

---

1 Taint analysis is used for bi-directional slicing, inter-slice dependency analysis, and handling asynchronous events as we describe in [3]
Chapter 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 3.1 illustrates the three components.

Network-aware program slicing\(^1\) 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 formats and signatures for each. 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 performs analysis using 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 repurposes FlowDroid’s static taint analysis and

\(^1\)Hyunwoo Choi contributed to the program slicing module.
\(^2\)We refer to objects as network I/O bound if they either originate from network messages or eventually go out to the network.
public Boolean doInBackground(Void... zzz) {
    HttpEntity entity = null;
    InputStream in;
    String url;
    StringBuilder sb;
    if (Constants.FRONTPAGE_STRING.equals(mSubreddit)) {
        sb = new StringBuilder(Cons

Figure 3.2: Reddit Client (Diode)'s Request & Response slice example

uses it 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 with reasonable 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 to identify
all messages because generic objects (e.g., array) can also be network-I/O bound.

Instead, our main idea is to start from network access methods and taint network buffers (e.g., socket.
getOutputStream() and socket.getInputStream()). If we taint these objects and perform taint prop-
agation, then we would be able to track all objects that read from input or write to output buffers.

Extractocol applies this idea to HTTP processing. For HTTP, Android apps typically use HttpServletRequest
and HttpResponse objects as input and output buffers. Figure 3.2 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 3.2, 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.

**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 and adds all statements that include tainted objects into the program slice, during taint propagation. We use FlowDroid’s standard rules for forward taint propagation. For backward propagation, Extractocol flips the edge direction of the control flow graph to inspect the statements in the reverse order and uses 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. Extractocol traces 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`, and `java.net`, for identifying HTTP-related objects, five 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 find that they are sufficient for modeling many 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 request 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.

\[
\begin{align*}
\text{sig}_{\text{pat}} & ::= \text{term} \mid \text{concat} (\text{term}, \text{term}) \\
& \quad \quad \mid \text{rep} (\text{term}) \mid \text{term} \lor \text{term} \\
\text{term} & ::= \text{constant} \mid \text{struct}_{\text{str}} \mid \top \mid \bot \\
\text{struct}_{\text{str}} & ::= \text{json} (\text{obj}) \mid \text{xml} (\text{obj}) \\
\text{obj} & ::= \text{key}_{\text{value}}^* \\
\text{key}_{\text{value}} & ::= (\text{key}, \text{value}) \\
\text{key} & ::= \text{constant} \\
\text{value} & ::= \text{constant} \mid \text{obj} \mid \text{array} \\
\text{constant} & ::= \text{num} \mid \text{integer} \mid \text{string} \\
\text{array} & ::= \text{value}^* \\
\end{align*}
\]

Figure 3.3: A simple language to represent signature patterns in Extractocol

2) Signature building: Extractocol inspects each of the three sub-slices to construct their signatures, using the same semantic model. It sets the earliest statement in a sub-slice as an entry point and processes each statement while traversing the data flow of the program slice. For each tainted object, it maintains a signature in the signature database. When it encounters a statement that updates tainted objects, such as request URIs, query strings, and text bodies, it keeps track of the string literals and objects being written to the object and updates their signatures according to its semantics. The signatures are represented using a simple language shown in Figure 3.3.

To follow inter-procedural data flows, Extractocol refers to inter-procedural control flow of the slice at a function-call or return statement and finds the next statement to inspect. It also performs flow-sensitive analysis within a procedure using a home-grown algorithm. Traditional data-flow analysis methods, such as the worklist algorithm, employ iterative analysis to obtain a fixed-point solution. For every change, it needs to revisit the basic blocks that may be influenced by the change. However, this leads excessive revisiting and scalability issues for Android applications, where we have to handle dense control flow graphs caused by implicit control transfers [19]. To be more scalable, we leverage the fact that our objective is to extract signatures for protocol messages, represented using a combination and repetition of string objects (Figure 3.3).

Within a procedure, it processes the statements in topological order of the intra-procedural control flow graph. It then recursively updates the signature using the following rules to handle confluence points. Extractocol maintains a signature database, which maps a variable to its signature, during the process. At a confluence point, it merges the signature database from predecessors. If all the instances of the same variable are well defined and the confluence point is not a loop header or latch, Extractocol merges them as a logical disjunction (\lor). If the confluence point is a loop header or latch, Extractocol identifies the
Algorithm 1: SigGen($S, sp, \Gamma$): Signature building

Data: The API semantic model, $D$, maps string operations and methods to corresponding rules

Input: Statements from a subslice, $S$, a starting point $sp$, a signature database $\Gamma = \{ v \rightarrow sig, \cdots \}$, which is a map from a string object to a signature pattern (of $sig_{pat}$ type)

Output: Processed signature database $\Gamma$

begin
Get a list of statements, $S_{sp}$, starting from $sp$ in topological order of control flow
foreach $stmt \in S_{sp}$ do
if $stmt$ is a confluence point then
Get all the signature databases, $\Gamma_i$ from all predecessor of $stmt$.
Merge all $\Gamma_i$ to $\Gamma$: $\Gamma = \bigsqcup \Gamma_i$
if $stmt$ has a string operation, $op \in D$, or calls a method, $m \in D$ then
$v \leftarrow$ get object($stmt$)
sig$_{op} \leftarrow D(op)$ (or $D(m)$)
Apply sig$_{op}$ to the signature database for $v$: $\Gamma(v) = sig_{op}(v)$
else if $stmt$ is a function call then
$f \leftarrow$ get procedure($stmt$)
$s \leftarrow$ get first stmt($f$)
$\Gamma \leftarrow$ SigGen($f$, $s$, $\Gamma$)
else
ignore stmt
return $\Gamma$

loop variant part of string objects and denotes it using $\text{rep}$ to mark the part can be repeated in the signature. Algorithm 1 describes the detailed intra-procedural signature building process, SigGen.

For a final output, Extractocol converts the signature language to a regular expression. For JSON and XML objects, the result are of tree structure whose leaves are string literals or numbers. The regular expression for the objects, thus, can be matched against valid variations. Repetitions ($\text{rep}$) and disjunctions ($\lor$) are respectively represented as the Kleene star and $|$ in regular expressions.

Note, for response bodies, our signatures may be partial in the general case. This is because the client may not necessarily process the whole message (e.g., server might send data that clients ignore). This limitation arises from the fact that we only analyze the client code. We claim that this is unavoidable, because server binaries are typically unavailable for most Android apps.

Example: Figure 3.2 shows an example code for Diode, a popular open-source browser for Reddit, on Google Play.
The network-aware program slicing of Extractocol effectively identifies the request/response slices. Among all the flows that pass a demarcation point, Extractocol extracts 6.3% of the code. (Figure 3.4 in Appendix shows all the flows related to the demarcation point in the inter-procedural control flow graph (ICFG) of Diode. The identified request/response slices are highlighted.)

Extractocol then builds signatures on the slices. From HttpClient.execute() (demarcation point) of Figure 3.2, 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,

\[ \text{Note that, although Extractocol analyzes Dalvik bytecode in an intermediate language, we show its source code for illustration purpose.} \]
Figure 3.4: A subgraph of ICFG of Diode on a demarcation point. The request/response slices are highlighted.

Extractocol outputs all the URI signatures as regular expressions. For example, http://www.reddit.com/search/.json?q=(.*)&sort=(.*) and its request method is GET. Figure 3.5 in Appendix shows all the output for URI with the control flow graph in basic blocks.

Figure 3.5: Signature outputs and control flow graph in basic block from the request slices of Diode

### 3.3 Message Dependency Analysis

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

**Pairing:** The basic idea for request-response pairing is to use information flow analysis to identify the response message body that depends on a given request URI. For this, Extractocol uses URI slices as taint source and response slice as sink and performs taint propagation described in §3.1. To be successful, the analysis must identify a one-to-one mapping between response and request. However, this 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 3.6 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 \texttt{requestA()} and \texttt{requestB()}, but soon reconverge to share a common demarcation point in \texttt{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 would discover paths to both A’s and B’s responses from A’s request (i.e., two paths exists from segment \([1]\) to \([6]\)).

Figure 3.6: Extractocol locates disjoint code segments, between which only a single path exists.

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. This method accurately identifies HTTP transactions, as we demonstrate in §5.

**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 Asynchronous Events

Asynchrony 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 [19]. FlowDroid, in particular, assumes an arbitrarily ordering of these events. This results in a failure to identify the dependencies across these events. For example, in the example above, the dependency between the first event (location service) and second event (user click) is lost. Thus, the part of query string created by the location service cannot be identified.

Note, this is also non-trivial for dynamic analysis because it is difficult to generate all asynchronous events dynamically (e.g., server message 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 all implicit data flows that potentially impacts the request. Our experience shows that, for commercial apps with rich UI, this dramatically improves the signature accuracy.
Chapter 4. Implementation

Our implementation consists of two modules: 1) an inter-procedural object dependency analysis that extends FlowDroid and Soot and 2) a signature generation module that performs semantic analysis.

The dependency analysis module consists of 2475 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 [47], a static analysis framework that provides the Jimple intermediate representation for Java and a call graph analysis framework [49]; Dexpler [20] that converts Dalvik bytecode to Jimple; and IFDS [59] 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 34 demarcation points in org.apache.http, android.net.http, java.net.HttpURLConnection, android.media.MediaPlayer, android.media.AsyncPlayer, javax.net.ssl, and java.net.URL from HTTP libraries [1]. We also support dependencies that occur between implicit call flows, such as intent and thread. For example, we track the messages passed via Intent that is often used in Android programming by inferring the receiver and sender of the intent.

The semantic analysis module consists of 5875 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. 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 six XML and JSON APIs, including org.json, com.google.gson, org.xml, etc.

Discussions: 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, and native binary. Extractocol can be easily extended to support some 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. Supporting JNI is more challenging because most tools perform dynamic analysis, and static analysis on native code is not mature enough for extracting protocols from realistic native binary [56]. One approach is to use dynamic analysis on native code [2] 58 to reconstruct the semantics of JNI and use them on Extractocol.

Finally, we are subject to limitations of existing tools that track program flows [19, 47]. On Android, discovering its implicit callback is an active area of research, and a number of studies [25, 55] are devoted to addressing the issue. Recently, EDGEMINER [25], identified 19,647 callbacks, many of which existing framework (e.g., Soot/FlowDroid) cannot handle. Although we add support for many popular implicit callbacks, obtaining a complete flow graph for Android apps requires further research [25, 55]. We leave this and large scale analysis as future work and focus on in-depth analysis and its methodology in this work.

1Note, many apps use these APIs directly or indirectly using third party libraries that use these APIs internally.
2Jimple 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.
Chapter 5. Evaluation

We perform in-depth case studies using twenty applications and demonstrate that Extractocol can provide detailed analysis on each individual application.

Our evaluation answers three key questions:
1. Does it produce accurate signatures with high coverage?
2. Does it effectively characterize app behaviors?
3. Can it reverse-engineer (private) REST APIs?

5.1 Validation of Protocol Analysis

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

Dataset: We use 14 apps from an open source app repository (F-Droid) [5] (twelve of them are also available on Google Play) and 5 commercial apps with 1 million+ downloads from 5 categories in Google Play. Table 5.1 summarizes their protocol features. For the open source apps, we obtain the ground truth by carefully inspecting the source code. For all applications, we collect traffic traces of all HTTP(S) transactions using manual UI-fuzzing, which often requires manual interventions, such as signing up and logging in for services. To capture and decrypt HTTPS messages, we use man-in-the-middle proxies [11, 15]. Note, some commercial apps are obfuscated. For open source apps, we obfuscate their APKs using ProGuard [16] and verify that the same results hold as non-obfuscated APKs.

<table>
<thead>
<tr>
<th>App</th>
<th>Protocol</th>
<th>Message Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adblock Plus (ADP)</td>
<td>HTTPS</td>
<td>GET, POST XML</td>
</tr>
<tr>
<td>AnarXiv (AXV)</td>
<td>HTTP</td>
<td>GET, XML</td>
</tr>
<tr>
<td>blippex (BLP)</td>
<td>HTTPS</td>
<td>GET, JSON</td>
</tr>
<tr>
<td>Diaspora WebClient (DW)</td>
<td>HTTP</td>
<td>GET, JSON</td>
</tr>
<tr>
<td>Diode (DIO)</td>
<td>HTTP(S)</td>
<td>GET, POST JSON</td>
</tr>
<tr>
<td>iFixit (FX)</td>
<td>HTTP</td>
<td>GET, JSON</td>
</tr>
<tr>
<td>Lightning (LTN)</td>
<td>HTTP(S)</td>
<td>GET, XML</td>
</tr>
<tr>
<td>qBittorrent (QBT)</td>
<td>HTTP</td>
<td>GET, POST JSON</td>
</tr>
<tr>
<td>radio reddit (RBD)</td>
<td>HTTP(S)</td>
<td>GET, POST JSON</td>
</tr>
<tr>
<td>Redditor (RDN)</td>
<td>HTTP(S)</td>
<td>GET, POST JSON</td>
</tr>
<tr>
<td>Twister (TWT)</td>
<td>HTTP</td>
<td>POST, JSON</td>
</tr>
<tr>
<td>TZM (TZM)</td>
<td>HTTPS</td>
<td>GET, XML</td>
</tr>
<tr>
<td>Wallabag (WLB)</td>
<td>HTTP</td>
<td>GET, XML</td>
</tr>
<tr>
<td>Weather Notification (WTN)</td>
<td>HTTP</td>
<td>GET, JSON</td>
</tr>
<tr>
<td>AOL: Mail, News &amp; Video (AOL)</td>
<td>HTTP(S)</td>
<td>GET, POST, XML</td>
</tr>
<tr>
<td>AC App for Android (ACA)</td>
<td>HTTP(S)</td>
<td>GET, POST, JSON, XML</td>
</tr>
<tr>
<td>AccuWeather (ACW)</td>
<td>HTTP</td>
<td>GET, JSON</td>
</tr>
<tr>
<td>KAYAK (KAY)</td>
<td>HTTPS</td>
<td>GET, POST JSON</td>
</tr>
<tr>
<td>TED (TED)</td>
<td>HTTPS</td>
<td>GET, POST JSON</td>
</tr>
</tbody>
</table>

Table 5.1: Open-source and commercial App (gray box) test case summary
Classifying Static And Dynamic URLs: For easy comparison, first we manually group the request URIs into unique patterns. URIs in each pattern share 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 transnational relationships match network traffics generated by PUMA.

Coverage: To verify the result with real network traces, we execute TED using PUMA, a state-of-the-art automated UI testing tool. PUMA took 10.3 minutes and Extractocol took 132.5 minutes to analyze all 19 apps. Note, PUMA merely generates traffic by executing all UI events, while Extractocol performs detailed protocol analysis.

Table 5.2 shows the number of HTTP messages identified by Extractocol by their request method and message type (e.g., request or response body) (e.g., Adblock Plus has two GET and one POST request, out of which one generates a response body). For open-source apps, we report the percentage of HTTP messages that Extractocol identified using the source code as ground truth. Whereas, Extractocol uses the request URIs and responses obtained from PUMA as ground truth in the case of commercial apps because we have not their source code.

<table>
<thead>
<tr>
<th>App</th>
<th>GET</th>
<th>POST</th>
<th>Body</th>
<th>Body</th>
<th>#Pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADP</td>
<td>2 (100%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>AXV</td>
<td>2 (100%)</td>
<td>-</td>
<td>-</td>
<td>2 (100%)</td>
<td>2 (100%)</td>
</tr>
<tr>
<td>BLP</td>
<td>1 (100%)</td>
<td>-</td>
<td>-</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>DIW</td>
<td>1 (100%)</td>
<td>-</td>
<td>-</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>DIO</td>
<td>24 (100%)</td>
<td>-</td>
<td>-</td>
<td>2 (100%)</td>
<td>5 (100%)</td>
</tr>
<tr>
<td>IFX</td>
<td>15 (100%)</td>
<td>7 (100%)</td>
<td>3 (100%)</td>
<td>14 (100%)</td>
<td>14 (100%)</td>
</tr>
<tr>
<td>LNT</td>
<td>2 (100%)</td>
<td>-</td>
<td>-</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>QBT</td>
<td>3 (100%)</td>
<td>13 (100%)</td>
<td>13 (100%)</td>
<td>3 (100%)</td>
<td>3 (100%)</td>
</tr>
<tr>
<td>RDI</td>
<td>3 (100%)</td>
<td>3 (100%)</td>
<td>3 (100%)</td>
<td>4 (100%)</td>
<td>4 (100%)</td>
</tr>
<tr>
<td>RDN</td>
<td>3 (100%)</td>
<td>3 (100%)</td>
<td>-</td>
<td>6 (100%)</td>
<td>6 (100%)</td>
</tr>
<tr>
<td>TWT</td>
<td>-</td>
<td>11 (100%)</td>
<td>11 (100%)</td>
<td>8 (100%)</td>
<td>8 (100%)</td>
</tr>
<tr>
<td>TZW</td>
<td>2 (100%)</td>
<td>-</td>
<td>-</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>WLB</td>
<td>1 (100%)</td>
<td>-</td>
<td>-</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>WTN</td>
<td>2 (100%)</td>
<td>-</td>
<td>-</td>
<td>2 (100%)</td>
<td>2 (100%)</td>
</tr>
<tr>
<td>AOL</td>
<td>3 (100%)</td>
<td>10 (66%)</td>
<td>10 (66%)</td>
<td>13 (66%)</td>
<td>13 (72%)</td>
</tr>
<tr>
<td>ACA</td>
<td>9 (100%)</td>
<td>15 (100%)</td>
<td>15 (100%)</td>
<td>23 (100%)</td>
<td>23 (100%)</td>
</tr>
<tr>
<td>ACW</td>
<td>15 (100%)</td>
<td>3 (100%)</td>
<td>3 (100%)</td>
<td>16 (100%)</td>
<td>16 (100%)</td>
</tr>
<tr>
<td>KYK</td>
<td>39 (100%)</td>
<td>7 (100%)</td>
<td>7 (100%)</td>
<td>6 (100%)</td>
<td>6 (100%)</td>
</tr>
<tr>
<td>TED</td>
<td>16 (100%)</td>
<td>2 (100%)</td>
<td>2 (100%)</td>
<td>10 (100%)</td>
<td>10 (100%)</td>
</tr>
</tbody>
</table>

Table 5.2: Coverage: # of request/response produced by Extractocol and % of identified signatures compared to the ground truth

The result shows that Extractocol generates all 339 message signatures and identifies all 119 HTTP (request URI-response body) pairs. Note, it only pairs responses that have bodies that are processed by the apps. Also, we verify that allmost all signatures and the pairs successfully match with our traffic traces.
Additionally, we discover interesting things which stem from the limitation of dynamic analysis and static analysis in commercial app cases. In TED case, Extractocol identifies 4 more transactions than PUMA does. For example, PUMA does not identify a request triggered in response to content updates, triggered by the server. Dynamic analysis is not able to reproduce such externally triggered events. In contrast, PUMA identifies 5 more transactions than Extractocol did for the AOL app (Table 5.3). We reverse-engineer AOL to determine the reason why Extractocol cannot extract the transactions. We reveal that AOL performs the HTTP operations through Intent which is a messaging object used to request an action from another app component. Many existing static analysis tools and Extractocol cannot properly consider Intent when they analyze android binaries. Therefore, although AOL hard-codes the URLs for the transactions in their binary (APK), Extractocol just represents them as wildcards (\.*). EPICC try to reduce the discovery of inter-component communication (ICC) in smartphones to an instance of the Interprocedural Distributive Environment (IDE) problem, and develop a sound static analysis technique targeted to the Android platform [55]. If Extractocol uses EPICC’s analysis results, it can build more meaningful signatures than Extractocol does for AOL app.

### Logical equivalence and signature validity

Table 5.3: The Summary URLs of AOL

<table>
<thead>
<tr>
<th>Signature</th>
<th>Real Traffic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(<a href="http://api.hd.aol.com/1/1/(.*)/appStart)(?app=)(.*)(&amp;codebase=)(">http://api.hd.aol.com/1/1/(.*)/appStart)(?app=)(.*)(&amp;codebase=)(</a>.*)</td>
<td><a href="https://applab-sdk.amazon.com/1.0/applications/0e29fb7721cf4814a8382312955ec303/events">https://applab-sdk.amazon.com/1.0/applications/0e29fb7721cf4814a8382312955ec303/events</a></td>
</tr>
<tr>
<td>(<a href="http://app.hd.aol.com/feed/videos?app=aolcom..https://cis.aol.com/cis/api/v1/GetArticleshttps://cis.aol.com/api/v1/CheckNewArticleshttps://cis.aol.com/api/v1/GetConfighttps://cis.aol.com/cis/api/v1/GetArticleDetails/">http://app.hd.aol.com/feed/videos?app=aolcom..https://cis.aol.com/cis/api/v1/GetArticleshttps://cis.aol.com/api/v1/CheckNewArticleshttps://cis.aol.com/api/v1/GetConfighttps://cis.aol.com/cis/api/v1/GetArticleDetails/</a>...)</td>
<td>Figure 5.1 (upper right) and Figure 5.2 (upper right) show the number of constant keywords identified by each signature. For example, in open-source signatures, for request body, a total of 145 keywords are...</td>
</tr>
</tbody>
</table>

We further verify whether our regex signature is logically equivalent to the source code and it matches with actual traffic traces. For open-source app URIs, we verify that all of them are logically equivalent to the source code. Surprisingly, the traffic traces from manual UI-fuzzing only contained 95 out of 98 URIs (Figure 5.1). All 95 URIs from traffic traces match with our signatures. Manual UI-fuzzing does not generate three requests that are not triggered by any human-generated events. One of them, for example, is an APK update request, triggered periodically by timer events. This shows that Extractocol can identify messages that UI-based fuzzing cannot. For commercial apps URIs, we verify that all of them are logically equivalent to manual UI-fuzzing results as well.

For request bodies and query strings in open-source apps, we identify 92 signatures, but manual UI-fuzzing misses one, which is the APK update response. All remaining 91 traces match with our regex signatures. All signatures are equivalent to the source code. Note, our heuristics in §3.4 identifies implicit dependencies by tracking across multiple asynchronous events. For example, in RRD, a JSON key-value pair string is generated from a user input and stored in a heap object. At a later time, another event triggers an HTTP request. Extractocol successfully handles this case.

Figure 5.1 (upper right) and Figure 5.2 (upper right) show the number of constant keywords identified by each signature. For example, in open-source signatures, for request body, a total of 145 keywords are...
identified through manual source code analysis and manual UI-fuzzing. Extractocol identifies all but one keyword is missed (one missing from the above-mentioned JSON object). We verify the response body signatures in the same way. We count the number of keywords in JSON and XML bodies (Figure 5.1 (right lower)). These include the keys in JSON key-value pairs and XML tags and attributes. It shows a different trend. Responses from traffic traces contain 616 keywords. However, Extractocol and manual source code analysis identify only 60% of them. This is because common apps often does not inspect all keywords received \(^3\). Despite this, all response body signatures are equivalent to the source code and generate a valid match with network traces. Note, these unused keywords and values do not affect the app protocol and the application semantics, unless the number of keywords or their order delivers an implicit message. Excluding these unlikely cases, Extractocol accurately characterizes the app behavior.

To further quantify the signature quality, we count the number of bytes that matches with our regex. \(R_k\) and \(R_v\) respectively denote the fraction of matched byte count on the constant keywords and the corresponding variable value parts of our signature. \(R_n\) is the fraction of byte count that is not matched by either of the two.

<table>
<thead>
<tr>
<th>URI ((R_k, R_v, R_n))</th>
<th>Request Body Query String ((R_k, R_v, R_n))</th>
<th>Response Body ((R_k, R_v, R_n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open-source apps 70/30/-%</td>
<td>47/52/1%</td>
<td>7/48/45%</td>
</tr>
<tr>
<td>Commercial apps 62/38/-%</td>
<td>71/29/0%</td>
<td>26/60/14%</td>
</tr>
</tbody>
</table>

Table 5.4: Matched byte count % on actual traffic

Table 5.4 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 62% bytes match in commercial apps. For query strings and request bodies, 99% of all bytes for open-source apps and 100% for commercial apps match with key-value pairs identified by Extractocol. For response body in open-source apps, only 55% of bytes match with them, and the rest is covered by wildcards in our regex because apps do not process all response data as noted earlier. However, for response body in commercial apps, 86% of all bytes matches them, this is due to special keywords (e.g., ‘description’ keyword with variable value parts of our regex) that have a large mount of bytes.
5.2 Characterization of App Behavior

To demonstrate Extractocol’s effectiveness in characterizing app’s protocol behavior, we perform detailed case studies using TED and radio reddit [12]. These apps use video and music streaming. TED is a popular app from the “Best Apps of 2014” category 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>
<thead>
<tr>
<th>#</th>
<th>Request (Static/Dynamically-derived URI)</th>
<th>Response</th>
<th>Dependency graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Speaker’s info (S)</td>
<td>JSON</td>
<td></td>
</tr>
<tr>
<td></td>
<td>name/description</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>inserted to DB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Facebook sharing (S)</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Advertisement query (S)</td>
<td>Ad query URI</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>GET (*): Ad query URI from #3 (D)</td>
<td>Ad resource URIs</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>GET (*): Ad video URI from #4 (D)</td>
<td>Binary</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Talk info (S)</td>
<td>thumbnail/video URIs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>inserted to DB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>GET (*): Thumbnail URI from DB (D)</td>
<td>Binary</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>GET (*): Audio/video URI from DB (D)</td>
<td>Binary</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.5: Selected HTTP transactions and its dependency graph for TED

Table 5.5 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.Resources class. 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.

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 5.6 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
Table 5.7: (Radio) reddit [7, 8, 53] API document

<table>
<thead>
<tr>
<th>Source</th>
<th>HTTP Method &amp; URI</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>hidden</td>
<td>GET reddit.com/api/info.json</td>
<td>-</td>
</tr>
<tr>
<td>radiodd</td>
<td>GET radioreddit.com/api/status.json</td>
<td>-</td>
</tr>
<tr>
<td>reddit</td>
<td>POST reddit.com/api/login</td>
<td>api_type, passwd, user, rem (optional)</td>
</tr>
<tr>
<td>reddit</td>
<td>POST reddit.com/api/save</td>
<td>category, id, uh</td>
</tr>
<tr>
<td>reddit</td>
<td>POST reddit.com/api/unsave</td>
<td>id, uh</td>
</tr>
<tr>
<td>reddit</td>
<td>POST reddit.com/api/vote</td>
<td>dir, id, uh, v (optional)</td>
</tr>
</tbody>
</table>

Table 5.8: Traffic trace for RRD transaction #2

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(\u2026)songs :: Surviving Minds", "reddit_url": ".(omitted), "reddit": "song", "score": ":6", "title": "Surviving Minds" } } }
```

5.3 Reverse-Engineering

Manual reverse-engineering attempts use packet traces to analyze popular REST APIs [14, 17, 33, 34, 60, 67]. 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 [14], and a repurposed private API was introduced. We compare our analysis results with a manual analysis result in [14], 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 [14]. Additionally, Extractocol reveals 14 times more APIs in just 31 minutes. It also
<table>
<thead>
<tr>
<th>Cat.</th>
<th>Method</th>
<th>URI Prefix</th>
<th># APIs</th>
<th>Example Sub URI</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Travel Planner</td>
<td>GET</td>
<td><a href="https://www.kayak.com/trips/v2">https://www.kayak.com/trips/v2</a></td>
<td>11</td>
<td>/edit/trip/</td>
<td>-</td>
</tr>
<tr>
<td>Authentication</td>
<td>POST</td>
<td><a href="https://www.kayak.com/k/authajax">https://www.kayak.com/k/authajax</a></td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Facebook Auth</td>
<td>POST</td>
<td><a href="https://www.kayak.com/k/run/fbauth">https://www.kayak.com/k/run/fbauth</a></td>
<td>2</td>
<td>/login</td>
<td>-</td>
</tr>
<tr>
<td>Flight</td>
<td>GET</td>
<td><a href="https://www.kayak.com/api/search/V8/">https://www.kayak.com/api/search/V8/</a></td>
<td>6</td>
<td>/flight/start</td>
<td>JSON</td>
</tr>
<tr>
<td>Hotel</td>
<td>GET</td>
<td><a href="https://www.kayak.com/api/search/V8/">https://www.kayak.com/api/search/V8/</a></td>
<td>2</td>
<td>/hotel/detail</td>
<td>JSON</td>
</tr>
<tr>
<td>Car</td>
<td>GET</td>
<td><a href="https://www.kayak.com/api/search/V8/">https://www.kayak.com/api/search/V8/</a></td>
<td>1</td>
<td>/car/poll</td>
<td>JSON</td>
</tr>
<tr>
<td>Mobile Specific</td>
<td>GET</td>
<td><a href="https://www.kayak.com/k/mobileapis">https://www.kayak.com/k/mobileapis</a></td>
<td>12</td>
<td>/currency/allRates</td>
<td>JSON</td>
</tr>
<tr>
<td>Advertising</td>
<td>GET</td>
<td><a href="https://www.kayak.com/s/mobileads">https://www.kayak.com/s/mobileads</a></td>
<td>1</td>
<td>-</td>
<td>JSON</td>
</tr>
<tr>
<td>Etc.</td>
<td>POST</td>
<td><a href="https://www.kayak.com/k">https://www.kayak.com/k</a></td>
<td>4</td>
<td>/cookie</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.9: A summary of Kayak API analysis results

identifies the use of app-specific HTTP header, “User-Agent : kayakandroidphone/8.1”. Table 5.9 shows a summary of requests grouped into eight categories by their URI prefix. Table 5.10 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 5.10) 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 [14] 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.

### Table 5.10: Selected Request Signatures for Kayak

<table>
<thead>
<tr>
<th>Sub URIs</th>
<th>Query String</th>
</tr>
</thead>
<tbody>
<tr>
<td>/k/authajax</td>
<td>action=registerandroid&amp;uuid=.<em>&amp;hash=.</em>&amp;model=.* &amp;platform=android&amp;cos=.<em>&amp;locale=.</em>&amp;tz=.*</td>
</tr>
<tr>
<td>/api/search</td>
<td>cabin=.<em>&amp;travelers=.</em>&amp;origin=.<em>&amp;nearbyO=.</em>&amp;destination=.*</td>
</tr>
<tr>
<td>/V8/flight/start</td>
<td>&amp;nearbyD=.<em>&amp;depart_date=.</em>&amp;depart_time=.* &amp;depart_date_flex=.* &amp;sid=.*</td>
</tr>
<tr>
<td>/api/search</td>
<td>searchid=.<em>&amp;nc=.</em>&amp;c=.<em>&amp;k=.</em>&amp;kde=up&amp;currency=.* &amp;includeopques=true &amp;includeSplit=false</td>
</tr>
</tbody>
</table>

Table 5.10: Selected Request Signatures for Kayak
Chapter 6. Related work

This work builds on a large body of work that performs automated program analysis for Android applications.

**Android program analysis:** Prior work on Android program analysis focuses on discovering privacy-sensitive information leakage [19, 26, 37, 41, 42, 45, 51, 55, 65, 70, 71] or identifying application misbehaviors [36, 44, 61]. Taint analysis tracks information flows to reveal unintended information leakage. TaintDroid [37] 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. Dynamic analysis can be fooled by malicious apps that act benign when they recognized they are being analyzed [37, 41].

To overcome this challenge, many studies employ static analysis [19, 41, 70]. These studies commonly reconstruct ICFG by modeling Android app’s lifecycle. 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 [19] to reconstruct the ICFG and use data dependencies in three different ways for protocol analysis. Finally, CryptoLint [56] detects the misuse of cryptographic libraries using static program slicing. SMV-Hunter [61] identifies Android applications that fail to properly validate SSL certificates.

**Fingerprinting Android app traffic:** FLOWR [68, 69] tries to classify mobile application traffic by extracting key-value pairs from HTTP sessions. NetworkProfiler [32] uses UI-based fuzzing on Android apps to build a comprehensive network trace. The main focus of these studies is to generate network traffic and extract unique finger-prints, rather than protocol analysis.

Previous studies [24, 28, 29, 31, 66] have focused on reverse-engineering general application protocols, such as HTTP, SIP, DNS, IRC, and SMB. Their approach can be categorized into two: one that only looks at network traces to infer the protocol and the one that uses execution traces of the program along with network traces.

**Protocol analysis using network traces:** Many studies [18, 21, 29, 30, 40, 46, 48, 63, 64, 68] use traffic traces as input to derive application protocol information, such as protocol syntax and state machine. Discoverer [29] infers message format to derive application protocol syntax, and ASAP [46] uses machine learning to extract typical communication signatures and their semantics. RolePlayer [30] and ScripGen [48] 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 [22] than program semantic analysis and cannot handle encrypted messages. Also, a common limitation of the approach is that obtaining a sufficient size of input that exhaustively contains protocol messages is hard especially due to highly skewed message popularity [29].

**Protocol analysis using program analysis:** Program analysis approach [22, 24, 28, 31, 50, 66] predominantly uses dynamic analysis as primary means to extract protocol information with different goals. Dispatcher [22] reverse engineers a botnet’s command-and-control (C&C) protocol to actively rewrite C&C messages. Replayer [54] and Rosetta [23] focus on replaying application dialogs, while
others [24, 28, 31, 50] aim for identifying protocol fields within a message. Prospex [28] 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); 3) require the input messages to exhaustively cover all message types; 4) require multiple messages instances per type for inferring general protocol format [28, 31, 50, 66]; and 5) rely on pattern inference for reconstructing client protocol state machine. In contrast, Extractocol provides richer behavioral information: 1) it analyzes both received and sent messages without server binary; 2) directly infers the message dependency (i.e., client protocol state machine); 3) tracks fine-grained relationship between request and responses across HTTP transactions; and 4) identifies how network messages are used. To our best knowledge, Extractocol is the first to provide such in-depth behavior analysis.

Recent work, such as Replayer [54] and SPA [56], uses static program analysis for protocol replay and automated interoperability testing of protocol implementations. Replayer [54] uses formal verification techniques and symbolic execution. However, the approach suffers from significant scalability issues [66] and falls short in demonstrating the usefulness in real-world applications. SPA [56] focuses on automatically testing protocol implementation with manually annotated source code.

---

1Existing work falls short in analyzing two-way communication. For most Android apps, the server binary is often unavailable.
Chapter 7. Conclusions

This work presents Extractocol, a framework for analyzing HTTP(S)-based application protocol behaviors for Android applications. 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 rich features including signature extraction, request-response pairing, and inter-transactional dependency analysis that are valuable in practice. Our in-depth evaluation on open-source and commercial 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.
References


Summary

Enabling Automatic Protocol Behavior Analysis for Android Applications

오늘날 안드로이드에서 발생되는 인터넷 트래픽은 40-50%에 달다고 있다. 네트워크 메니지먼트측면에서 애플리케이션의 네트워크 행동패턴을 분석하는 일은 매우 중요 하지만 애플리케이션의 행동패턴을 매우 자세히 분석하는 일은 어렵다. 최근 많은 수의 애플리케이션들이 HTTP(S) 프로토콜을 이용하고 JSON이나 XML 같은 데이터구조를 이용하여 통신하는 설정인데 이 때문에 애플리케이션들의 행동패턴을 파악하는 일은 더욱 어렵게 되었다. 이 연구에서는 자동적으로 애플리케이션의 네트워크 행동패턴을 분석하는 프레임워크(Extractocol)을 개발하였다. Extractocol은 안드로이드 애플리케이션의 실행파일 만을 입력 받아 HTTP 트랜잭션을 정확히 추론하고 각 HTTP 메세지의 포맷과 그들의 의존성 관계를 파악할 수 있다. 우리는 오픈소스 애플리케이션 및 상용 애플리케이션에 대한 다양한 실험을 수행하였다. 이를 통해 Extractocol 이 높은 커버리를 가진 장점에 대하여 정확한 애플리케이션의 행동 패턴을 추론할 수 있음을 보였다.
감 사 의 글

부족한 나에게 연구가 무엇인지 깨닫게 해 준 한동수 교수님, 항상 나를 위해서 살신성인하시서 위대하고 강한 나의 어머니 강미연 여사 두 분께 진심으로 감사의 말씀을 드린다.

마지막으로, 앞들었던 청소년기 채임만이 유일한 삶의 낙마였던 나에게 어떻게 사는 것이 맞겠게 사는 인생인지 알려주고 편구처럼 편안 아버지처럼 나의 삶에 지대한 영향을 가져 사람이다. 내가 한국 과학기술원에 입학할 수 있었던 것은 모두 그의 가르침 덕분이다. 지금은 연락이 닦질 않지만, 예전에 했던 말했던 언젠가 콧 다시 만난 것이라 믿고 있다. 내가 가장 존경하는 사람 송호 형에게 가장 큰 감사의 마음을 전한다.
이력서

이름: 김정민
E-mail 주소: appff@kaist.ac.kr

학력
2003. 3. – 2006. 2. 통영고등학교 (졸업)
2006. 3. – 2009. 2. 영남대학교 컴퓨터정보계열 (A.S.)
2009. 3. – 2014. 2. 영남대학교 컴퓨터공학과 (B.S.)
2014. 3. – 2016. 2. 한국과학기술원 정보보호대학원 (M.S.)

경력
2010. 1. – 2012. 12. (주)컴퓨터에이트 프로그래머
2015. 3. – 2015. 8. 한국과학기술원 정보보호대학원 사이버공격방어실습 I 조교
2015. 12. – 임운과학기술장학재단 11기 장학생

연구 업적
1. Hyunwoo Choi, Jeongmin Kim, Hyunwook Hong, Yongdae Kim, Jonghyup Lee, and Dongsu Han
   *Extractocol: Automatic Extraction of Application-level Protocol Behaviors for Android Applications, SIGCOMM’15 (Poster).*