# Mass Discovery of Android Traffic Imprints through Instantiated Partial Execution

Yi Chen[1,3], Wei You[2], Yeonjoon Lee[2], Kai Chen[1,3*], XiaoFeng Wang[2*], Wei Zou[1,3]

[1]{SKLOIS[§], CAS-KLONAT[†], BKLONSPT[‡]}, Institute of Information Engineering, Chinese Academy of Sciences
[2]School of Informatics and Computing, Indiana University Bloomington
[3]School of Cyber Security, University of Chinese Academy of Sciences
{chenyi,chenkai,zouwei}@iie.ac.cn,{youwei,yl52,xw7}@indiana.edu

## ABSTRACT

Monitoring network behaviors of mobile applications, controlling their resource access and detecting potentially harmful apps are becoming increasingly important for the security protection within today's organizational, ISP and carriers. For this purpose, apps need to be identified from their communication, based upon their individual traffic signatures (called *imprints* in our research). Creating imprints for a large number of apps is nontrivial, due to the challenges in comprehensively analyzing their network activities at a large scale, for millions of apps on today's rapidly-growing app marketplaces. Prior research relies on automatic exploration of an app's user interfaces (UIs) to trigger its network activities, which is less likely to scale given the cost of the operation (at least 5 minutes per app) and its effectiveness (limited coverage of an app's behaviors).

In this paper, we present *Tiger* (Traffic Imprint Generator), a novel technique that makes comprehensive app imprint generation possible in a massive scale. At the center of Tiger is a unique *instantiated slicing* technique, which aggressively prunes the program slice extracted from the app's network-related code by evaluating each variable's impact on possible network invariants, and removing those unlikely to contribute through assigning them concrete values. In this way, Tiger avoids exploring a large number of program paths unrelated to the app's identifiable traffic, thereby reducing the cost of the code analysis by more than one order of magnitude, in comparison with the conventional slicing and execution approach. Our experiments show that Tiger is capable of recovering an app's full network activities within 18 seconds, achieving over 98% coverage of its identifiable packets and 0.742% false detection rate on app identification. Further running the technique on over 200,000 real-world Android apps (including 78.23% potentially harmful apps) leads to the discovery of surprising new types of traffic invariants, including fake device information, hardcoded time values, session IDs and credentials, as well as complicated trigger conditions for an app's network activities, such as human involvement, Intent trigger and server-side instructions. Our findings demonstrate that many network activities cannot easily be invoked through automatic UI exploration and code-analysis based approaches present a promising alternative.

## CCS CONCEPTS

**•Security and privacy → Mobile and wireless security;**

## KEYWORDS

traffic signature, ISP, large scale, slicing, partial execution

## 1 INTRODUCTION

The pervasiveness of mobile devices mounts great pressure on today's network security infrastructures. Just like other desktop or web applications, mobile apps are supposed to be under the monitoring and protection of the security systems within enterprise, ISP or carriers. Particularly, with the threat of potentially-harmful apps (PHAs) [18] [10] [11] on the rise, there is a strong demand for detecting them at the network level, using the anti-malware systems deployed by individual organizations or mobile carriers (through their Managed Security Services [16]). Even for legitimate apps, those running on the personal devices brought by employees to their work places (dubbed "bring your own device" or BYOD) are increasingly required to be subject to the control of next-generation firewalls (NGFW), for managing their access to corporate resources (like bandwidth, internal servers, etc.) [17]. To serve these purposes, most important here is identification of individual apps from their communication traffic, before the PHA detection and access control can happen. This, however, is by no means trivial. Different from desktop applications, mobile apps typically are hard to be identified by their protocols and port numbers. Instead, they mainly use HTTP protocol with their port numbers changing continuously for every HTTP packet and rarely include app names within the `User-Agent` header as recommended for identification. Also looking for the servers the apps talk to (within the `HOST` field of the HTTP header) does not work either, simply because increasingly the same host (often belonging to third parties) serves many different apps (e.g., Baidu Map SDK [4] for location services). Therefore, effective techniques are needed to fingerprint individual apps from

[*] Corresponding Authors.
[§] State Key Laboratory of Information Security, IIE, CAS.
[†] Key Laboratory of Network Assessment Technology, CAS.
[‡] Beijing Key Laboratory of Network Security and Protection Technology

.

*CCS'17, Oct. 30–Nov. 3, 2017, Dallas, Texas, USA*
© 2017 ACM. ISBN 978-1-4503-4946-8/17/10…$15.00
DOI: http://dx.doi.org/0.1145/3133956.3134009

their network traffic, extracting their signatures (also called *imprints* in our study) from their packets at a large scale, given that already tens of millions of apps are in the wild.

**Imprint mass production: challenges**. Note that app traffic imprints cannot be produced at a large scale through manual analysis, as done by some NGFW companies today, which is in no position to handle any substantial portion of legitimate apps on various app markets, not even hundreds of thousands of PHAs reported by VirusTotal [4]. There are attempts to automate this process, for example, through analyzing app traffic traces recorded by an ISP to recover invariable and also distinguishing tokens for individual apps [28]. A question is for a given app, how to systematically generate such traffic before the invariants can be extracted. To this end, large NGFW providers like Palo Alto Networks utilize dynamic analysis, running individual apps and trying to trigger their network activities [31]. Techniques have been proposed to fuzz an app's user interfaces (UI) and enhance random testing tools such as Monkey [3] for a more targeted exploration of different UI paths. However, these techniques are less suitable for mass production of precise app imprints, due to fundamental limitations of dynamic analysis and UI fuzzing, which are hard to discover all network activities the users can possibly trigger. Further, such analysis takes time: actually, no one knows today how long an app needs to run in order to disclose all its network activities or when it has indeed been extensively explored [14] [13]. No techniques are available to enable a truly large scale and comprehensive analysis of app traffic for generating their network imprints.

**Tiger**. To address these challenges, we present a new technique to support a large-scale discovery of Android app imprints. Our system, called *Tiger* (Traffic Imprint Generator), is built upon a highly efficient invariant recovery mechanism that only partially executes an app under test, focusing on its code fragments directly contributing to the formation of the invariable tokens within the app's traffic. To this end, Tiger employs a novel backward slicing technique to work on every network API discovered in the app's code. Unlike a conventional slicing algorithm, our technique continuously evaluates every program statement related to the network API to discover variables which are unlikely to affect any constant value of the API's output (network packets). Once found, such a variable is immediately instantiated by assigning it an appropriate concrete value so as to avoid further backtracking other statements upstream that may have impacts on the variable. Essentially, this approach automatically *prunes* the code slice for each network sink, leaving only a small set of statements believed to contribute to the creation of invariants (e.g., a special URL, a key-value pair, a hidden app ID used by the developer) on the app's traffic, thereby cutting down the cost for code analysis and dynamically executing the slice. To further reduce the analysis complexity, Tiger also identifies shared code across different slices and replaces their output with concrete values generated in previous analysis. In this way, only such a highly simplified slice needs to be executed for recovering traffic tokens, which are then compared across those produced by other apps to form the app's unique traffic imprints.

This *instantiated partial execution* (IPE) technique is found to be very effective: in our evaluation, the IPE outperformed the conventional slicing and execution approach by 12.42 times (Section 4.3). On average, a commercial app was analyzed within 18 seconds with all its network sinks fully covered. Compared with prior approaches (e.g., [14]), which dynamically recover traffic signatures for app fingerprinting, Tiger produces richer traffic imprints, not only package names, advertising identifiers (Ad-IDs) and URL, but also keys and values extracted from HTTP headers and content (Section 5.2). Particularly, when running our approach against a prominent app usage identifer that generates easy-to-obtain signatures [39], Tiger discovered 43.98% more packets (Section 4.2) containing imprints. Further our study shows that among all the packets with identifiable traffic tokens, our approach achieved a coverage of 98.54%. This is important since mobile users tend to migrate across different networks (LTE and Wi-Fi), causing an organizational NGFW to miss some identifiable packets; therefore, the more packets an app's imprints can cover, the more likely it can be timely identified for detection and access control.

**Our findings**. The high efficiency achieved by Tiger enabled us to discover app traffic imprints at an unprecedented scale: we ran our system on over 200,000 apps including 78.23% PHAs from VirusTotal. Altogether, 392,645 invariants were discovered, uniquely characterizing all identifiable apps. Interestingly, some of these apps produce new types of invariable tokens never reported in prior research, including new keys and values within HTTP headers and content. Particularly, we found that shared libraries send out fake device information when their host apps do not have permissions to do so on mobile devices, and some apps communicate with their servers using hardcoded time values, credentials and their developers' personal information and even fixed session and cookie IDs, which were all automatically recovered as the apps' unique traffic identifiers. Further we discovered that indeed some traffic flows involving distinguishing imprints cannot be easily triggered: oftentimes human interventions are expected to get through login pages, and most intriguingly identifiable traffic could show up only when the Activity of an app is invoked by other apps or by the click on the URL scheme posted by its server to the app's webview. We also found that suspicious behaviors containing identifiable imprints can only be triggered by some special conditions (e.g., a certain event or a remote command). The findings strongly indicate that code-analysis based app fingerprinting is a promising alternative to automatic UI exploration, which is both slow (at least 5 minutes per app) and incapable of triggering the apps' real-world network activities in a comprehensive way.

**Contributions**. The contributions of the paper are summarized as follows:

- *Innovative traffic imprint generation.* We present a new technique that efficiently triggers an app's network activities and discovers its invariable tokens. Our approach utilizes a highly-efficient partial execution technique, which targets potential invariants and significantly simplifies the code slices for a network sink by instantiating the variables believed to be unrelated to the invariants. This allows us to quickly identify the invariable tokens for imprint generation. With the IPE mechanism, Tiger achieves over 98% coverage of identifiable packets with only 0.74% false detection rate. Not only does it outperform existing traffic fingerprinting techniques (for Android apps) but it runs at least an order of magnitude faster than the conventional slicing and execution, which is critically important for analyzing app traffic at a large scale.

• *Large-scale imprint discovery.* Running our technique on over 200,000 recent apps, the largest study of this kind, our research sheds new light on the traffic features of Android apps, including surprising traffic content that can serve as apps' unique traffic imprints and complicated conditions for triggering their identifiable network activities. The findings help us better understand apps' network behaviors that can uniquely characterize them, highlighting the code-analysis based approaches as a promising way to fingerprint network traffic of apps.

## 2 BACKGROUND

In this section, we discuss the traffic invariants of Android apps and explicate the assumptions made in our study.

**Traffic imprints of Android apps**. Most mobile apps are web applications, which operate through interacting with their server-side components. For example, news apps communicate with their servers for requesting latest news; map and weather utilities use geo-locations to retrieve from their servers pieces of map images or weather conditions. These activities leave invariable traffic content that could potentially allow apps identification. In this study, we refer to any field with invariable content (e.g., IP address, host name, key, value, Ad-ID and their transformations) in network flows as a token. Once a token or combination of several tokens in one single packet can uniquely identify an app, we call the combination an imprint.

The coverage of imprints is important to app access management and PHA detection. Like desktop or web programs, mobile apps are also expected to be monitored and controlled within individual organizations or by ISPs and carriers. Serving this purpose is the NGFW technologies, through which a firewall uses imprints to identify apps for capturing harmful code or enforcing security policies on their access to corporate resources. Note that this is very different from how traditional firewalls work, which largely rely on IPs and ports to find the targets they are meant to control. The approach cannot be applied to apps, since most of them use HTTP with their port numbers changing continuously for every HTTP packet. And therefore cannot be differentiated from each other according to their port numbers. For these apps, their imprints (also referred to as signature, fingerprints in prior studies) are the key to determining their presence.

Such imprints are discovered today through dynamic analysis, from the traffic flows generated by running individual apps, as all major NGFW providers (e.g., Palo Alto Networks, Dell, HP, Huawei, Sophos, MobileIron, etc) do. Various techniques have been developed for this purpose, for example, systematically fuzzing an app's user-interfaces. However, as mentioned earlier, these approaches cannot comprehensively explore an app's behavior, triggering all its network-related activities: for example, an app may require user login and a mobile game may need deep human involvements before moving to the stage where characterizing traffic is generated. Even more challenging is PHAs, which may include carefully-crafted conditions (such as time, locations, events, etc.) for hiding its malicious activities that cannot be easily triggered. Further, dynamic analysis alone tends to be heavyweight, less suitable for processing a large number of apps (on the order of hundreds of thousands or even millions). Development of effective techniques to make the imprint discovery more comprehensive and highly scalable is the aim of our research (Section 3).

**Assumptions**. Our study focuses on the apps capable of producing network traffic that carries identifiable imprints. A small portion of real-world apps do not generate traffic at all or do not have unique tokens in their communication flows, which are outside the scope of our study. Also we do not consider the traffic tokens introduced by an app's server-side logic, since these tokens could be altered by the server. Further, given the limitations of today's NGFW, which does not work well on stateful traffic signatures, all the imprints generated in our research are combinations of the invariable tokens within a single packet. Finally, it is important to note that the objective of our study is to generate imprints to serve network-based PHA detection and app management, which are widely deployed within organizations today and become increasingly important to ISPs and carriers. Our imprint generation approach is based on code analysis and independent from the network protocol (HTTP or HTTPS) adopted by apps. In this study, we focus on HTTP traffic as a first step, since the overwhelming majority of apps are HTTP-based [14]. For HTTPS traffic, a trusted HTTPS proxy need to be placed MITM to scan the cleartext of the traffic for matching with the generated imprints [33].

## 3 FINDING IMPRINTS WITH TIGER

Here we elaborate the design and implementation of Tiger, starting with a high-level description of the idea and the architecture of our system and then coming to technical details, particularly those of the IPE technique.

### 3.1 Overview

As mentioned earlier, Tiger is designed for a comprehensive, scalable analysis of Android apps, automatically identifying invariable tokens in the apps' network traffic. The key idea here is to statically locate all network sinks within an app, and then slice and prune the program to identify *the statements related to putative invariants* before partially executing the simplified slice to generate the invariant tokens if they indeed exist. To serve this purpose, Tiger includes a suite of technical innovations: it first run a coarse (yet fast) slicing to establish a relation between a statement (more precisely its variables) and a set of internal sources of invariants (e.g., constant values, manifest or other resource content); among such statements, it further performs a differential analysis to find the variables whose values have no impact on possible invariable tokens. By assigning concrete values (instantiation) to such variables, the IPE avoids the cost for back-tracking the statements affecting these variables, and also produces a simplified slice (with fewer statements) that can be quickly executed to obtain the tokens for constructing the app's traffic imprints.

**Architecture**. Figure 1 illustrates the architecture of Tiger, which includes the pre-processing module, IPE engine (with coarse slicing, differential analysis and cross-slice optimization), partial execution module and imprint generator. The pre-processor disassembles an app, locates all its network sinks (the APIs for sending messages) and builds up call graphs (CG) of the app. The IPE performs

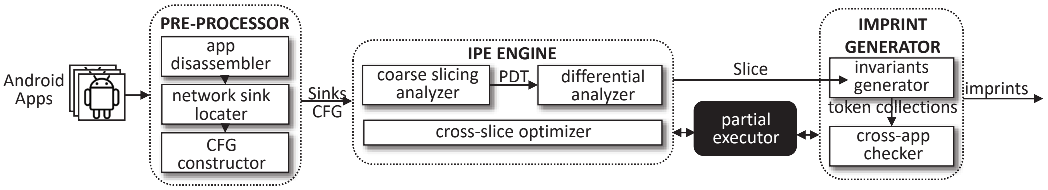
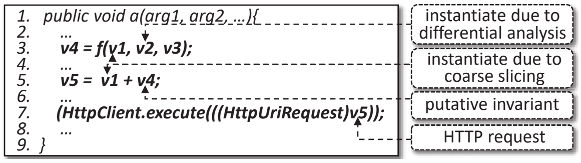


**Figure 1: Architecture of Tiger.**



```
1.  public void a(arg1, arg2, ...){
2.     ...
3.     v4 = f(v1, v2, v3);
4.     ...
5.     v5 =  v1 + v4;
6.     ...
7.     (HttpClient.execute(((HttpUriRequest)v5));
8.     ...
9.  }
```

instantiate due to differential analysis
instantiate due to coarse slicing
putative invariant
HTTP request

**Figure 2: An example showing how instantiated slicing finds a token.**

both coarse slicing and differential analysis for each sink, selects the invariant-related statements and runs these statements to acquire invariable tokens through the partial execution module. The generator constructs an imprint for the app using all the tokens found.

**An example**. Here we use an example (Figure 2) to walk through the whole imprint discovery process. The code fragment in the figure is from a popular Chinese app `com.tjsinfo.mangguoVideo`. Once the sink statement at Line 7 is discovered by the pre-processor, the IPE starts working on the variable $v5$ which represents an object of HTTP request, slicing the program backward until the statement at Line 5 with variables $v1$ and $v4$ encountered. So we need to check whether any variable among $v1$ and $v4$ affects $v5$. After running a coarse slicing (at the method level) on both variables, $v1$ is determined to be unrelated to any invariant source within the app while $v4$ does. So we instantiate $v1$, and further slice back on $v4$. We find that the content of $v4$, once adjusted, causes the change on a significant portion of the sink's output. That portion is considered to be a *putative* token. Going further up the control flow, the IPE finds that two new variables $v2$ and $v3$ at Line 3 impact $v4$ through function $f$. This triggers a differential analysis in which the IPE changes the content of $v2$ or $v3$ while fixing the other before running the partial slice toward the sink. The analysis result reveals that $v2$ does not contribute to the putative token. Hence, both $v1$ (unrelated to any invariable sources) and $v2$ (not affecting any invariable token) are assigned with concrete values, allowing the slicing to continue without exploring their corresponding branches. During this process, the IPE continuously checks the methods discovered along the CFG: if any of them shows up within a known slice (for a different sink within an app), the output of the method is instantiated based upon prior findings.

In the end, a simplified slice is constructed as illustrated in the figure in bold font. The slice is then executed to produce the output for the sink, which confirms that the putative output is indeed an invariant. This invariant, which is actually a collection of tokens (`data`, `nid`, `93535c6092f543e8a257ee435a69da06`), is further compared with those produced by other apps. If it is only present in the traffic of this app, the invariant is reported as one of the app's imprints. When a NGFW performs PHA detection/access control and finds a network packet containing the imprint, we say `com.tjsinfo.mangguoVideo` is identified. In the following we elaborate how these components work.

## 3.2 Instantiated Slicing

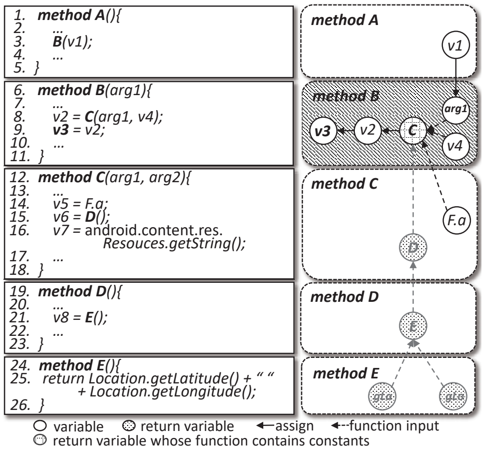
Critical to the high-performance design of Tiger is its IPE engine, which given a set of network sinks discovered by the preprocessor, quickly identifies the statements related to possible invariants produced by these sinks and further runs the statements to recover the tokens in the traffic. Serving this purpose is a unique *instantiated slicing* technique. More specifically, program slicing discovers a set of statements (called a *program slice*) that affect variables at some point of interest (the network sinks in this case). Execution of the slice leads to the disclosure of the variables' values. However, this conventional approach does not scale well on complicated programs, due to the difficulties in keeping track of potentially a large number of paths. To address these challenges, towards a highly efficient, scalable invariant discovery, our IPE engine is designed to prune within the sink-related paths, leaving only a small set of statements related to potential invariants. This is achieved through testing a variable's relevance to tokens, including a coarse slicing and differential analysis, and instantiating less essential variables to avoid exploring their branches. This approach enhances the performance of the slicing step by at least one order of magnitude, as observed Section 4.3, and achieves the accuracy (0.742% false detection rate and 98.54% app coverage, see Section 4.2). Following we first describe the pre-processing step that discovers the network sinks and then explicate our unique slicing technique.

**Pre-processing**. The preprocessor is designed to locate network sinks within an app and also convert the code to the form that can be easily analyzed by the IPE. Specifically, our implementation first disassembles the app's DEX bytecode to the SMALI[1] intermediate representation [35] and then builds a call graph (CG) across different methods within the app based on the idea from Flowdroid [8][2]. Then within each method involving network sinks (APIs) like org.apache.http.client.HttpClient.execute(), its control-flow graph (CFG) is created for the follow-up slicing analysis. Our approach further builds CFGs[3] for other methods whenever the backward slicing goes across the procedure boundary and enters these methods.

[1] We choose to use SMALI instead of a simpler IR (e.g., Jimple) since SMALI can represent the original bytecode in a more accurate way.

[2] Similar with Flowdroid, our CG construction approach takes into consideration the callback and message handling mechanisms. But currently, we do not support the resolution of reflective calls.

[3] We acknowledge that we do not handle ICC while building CFGs.



```
1.  method A(){
2.      ...
3.      B(v1);
4.      ...
5.  }
6.  method B(arg1){
7.      ...
8.      v2 = C(arg1, v4);
9.      v3 = v2;
10.     ...
11. }
12. method C(arg1, arg2){
13.     ...
14.     v5 = F.a;
15.     v6 = D();
16.     v7 = android.content.res.
                Resouces.getString();
17.     ...
18. }
19. method D(){
20.     ...
21.     v8 = E();
22.     ...
23. }
24. method E(){
25.  return Location.getLatitude() + " "
          + Location.getLongitude();
26. }
```

**Figure 3: Example of backward slicing. The initial slice is in method B, which is highlighted.**

**Backward slicing**. For each network sink, Tiger then conducts backward slicing to discover the statements affecting invariable tokens sent out through the sink. Specifically, starting from the input parameters of a sink API (e.g., $v$ in the API org.apache.http.client.HttpClient.execute(v)), our approach goes backward on the CFG of the method containing the sink (called *sink method*) to build an *initial slice* with all the statements within the method that directly or indirectly influence the sink. In the meantime, Tiger also constructs a *Possibly Dependent Tree* (PDT) for the method, with all variables potentially related to the output invariants. Specifically, for each statement on the slice, if it does not involve a function call, its variables are then added to the PDT. For example, in Figure 3, suppose that in `method B`, $v3$ is the content sent out at a network sink, and a statement on the slice is $v3 = v2$ at Line 9. Then $v2$ is added to the PDT with an edge from $v2$ to $v3$. Further Tiger slices back to Line 8. The statement assigns a return value of a function call to $v2$. So Tiger adds a node $C$ (which represents a return variable) and an edge from $C$ to $v2$. Tiger assumes that the returned variable is affected by all the inputs ($arg1$ and $v4$) of the function $C()$, therefore adds these variables to the PDT and labels them for a later revisit. Then our approach continues to slice the method until all its input variables are on the PDT. This process results in an initial slice as illustrated in the grayed box at the right-hand side of Figure 3, in the form of a *tree* rooted at the network sink.

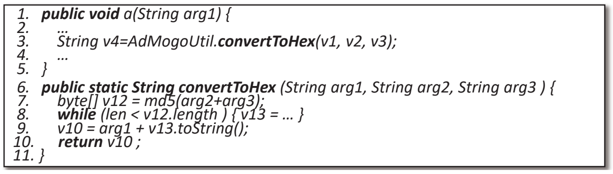
The initial slice is for a single method, the one containing the sink. The next step, naturally, is to *extend* the slice across procedures, to include related statements in the functions it calls and those in the methods that call it. Also, the variables on these statements (e.g., $v1$, $F.a$ and $D$ in Figure 3) need to be added to the PDT. The challenge here is the complexity of analyzing these methods, which requires not only slicing their code but traversing the CG to work on other methods they trigger or those that invoke them. Also stepping into such methods significantly raises the chance to encounter the program structures hard to process, loops in particular. Although techniques exist for estimating the number of iterations for each loop [40] or unwinding iterations [15], they are heavy-weight and less accurate. So our strategy is to avoid getting into these functions whenever possible, unless the variable received from a method (through parameters with which it calls the sink method) is considered highly likely to have an impact on the traffic invariants.

Specifically, Tiger is designed to strategically analyze the labeled variables from the app's initial slice, *only extending those likely to affect traffic tokens while instantiating others.* In this way, we can minimize the cost of inter-procedural slicing, remove all non-essential functions and their subtrees (the ones rooted at the function calls) and only execute the simplified slice to discover target network invariants. For this purpose, our approach performs two correlation analyses on each variable to identify irrelevant variables (which are instantiated later): coarse slicing and differential analysis. Note that our approach does not sacrifice accuracy. Tiger achieves over 98% coverage with 0.74% false detection rate (see Section 4.2).

**Sources of invariants and coarse slicing**. A key observation we leverage for irrelevant variable identification is that almost always, a traffic token produced by an app's network sink originates from some constant values within the program, such as constants, resource and manifest files. For example, package names appear in an app's manifest file; domain names are often hardcoded in the code or stored in the resource file. In our research, we found that all invariants observed from 100 randomly sampled apps exactly come from these sources. Therefore, we consider constant, manifest and resource files of an app as its *sources of invariants* and expect that invariable tokens in the app's traffic all have a data-flow related to these sources. This observation enables us to simplify the slicing operation by stopping the backward analysis on the variables unrelated to these sources. Specifically, for each variable encountered in PDT, Tiger quickly evaluates its contributions to the network invariants through a coarse slicing: for the variable returned from a function (including the return value and the field of a global object), our approach inspects the CG to extract a subgraph of calls invoked by the function and finds out whether any method on the subgraph contains a source of invariants; if none of them do, the variable is considered irrelevant. An example is that, in Figure 3, the return variable of $D$ does not need to be sliced.

This approach does not slice callees within a function. Instead it quickly goes through code statements of each callee and the callee's subgraph to find the constants and Android system APIs calls which access source of invariants (e.g., `android.os.Bundle.getStringArray()` for reading from the manifest and `android.content.res.Resources.getString()` for accessing to the resource files). Figure 3 presents an example. As we can see here, $v6$ in method $C()$ depends on the return value of method $D()$. Then coarse slicing quickly looks into the method $D()$ and its subgraph to check whether any source of invariants exists there. In this example, there only exist APIs for acquiring geo-location, which do not access any sources of invariants, indicating that $v6$ is irrelevant to the source and should be instantiated.

**Differential analysis**. Differential analysis is another technique for removing irrelevant variables from the slicing targets, when these variables cannot be identified through the coarse slicing. For a function that returns a variable in the PDT, the differential analysis identifies a subset of the function's *inputs* (any variables defined



```
1.  public void a(String arg1) {
2.      ...
3.      String v4=AdMogoUtil.convertToHex(v1, v2, v3);
4.      ...
5.  }
6.  public static String convertToHex (String arg1, String arg2, String arg3 ) {
7.      byte[] v12 = md5(arg2+arg3);
8.      while (len < v12.length ) { v13 = ... }
9.      v10 = arg1 + v13.toString();
10.     return v10 ;
11. }
```

**Figure 4: Example of differential analysis.**

outside the function before its use within the function, including the parameters from its caller, return variables from its internal calls and global variables) that do not contribute to the traffic token, and instantiates them with concrete values.

The whole idea of differential analysis comes from a key observation: given a function whose output is either a string or the content serializable to a string (e.g., `JSONObject`), if any of its inputs indeed contributes to the traffic token, the inputs must also affect the invariable part of this function's output, which serves to propagate invariable data from the inputs related to *sources of invariants* to the traffic invariant portion; in other words, any input independent of the function's invariable output will not relate to the sink's invariant. Based on this observation, our approach first identifies the invariable part of a function's output and utilizes a lightweight partial execution to test whether any input has an impact on this invariable portion. Specifically, consider a function with *n* inputs and a return variable *R*. Through the coarse slicing, Tiger is able to identify some of the inputs that do not contribute to the traffic token by checking their relations to the *sources of invariants*. Let $I_0, \cdots, I_k$ $(0 \leq k < n)$ be such *irrelevant* variables. Then we test whether any variable of $I_{k+1}, \cdots, I_n$ contributes to the invariable portion of *R*. Note that in the case that all inputs are irrelevant, *R* is instantiated directly as discussed before. Also, when none of the inputs can be dropped by the coarse slicing, we consider that the whole output *R* is invariable and each input of the function contributes to *R*.

To identify the invariable part in *R*, we randomly assign two sets of values to $I_0, \cdots, I_k$ and one set of values to the rest of inputs $I_{k+1}, \cdots, I_n$, according to their individual types. [4]. By executing the function (see Section 3.3) twice (each corresponding to a set of values for $I_1, \cdots, I_k$), two return values, $R_1$ and $R_2$, are produced, which are all strings, given that the output of the function we consider is either a string or the type that can be serialized to a string. The common substrings of $R_1$ and $R_2$ (denoted by $R_1 \cap R_2$), is then considered to be the invariable part of the function output. Then for the rest of the inputs $I_i \in \{I_{k+1}, ..., I_n\}$, Tiger tests every variable's contribution to the invariant by changing the value of $I_i$ while keeping the content of other variables intact, before running the function with the new inputs to get a new return string $R_3$. If $R_1 \cap R_2 = R_2 \cap R_3$ (that is, $R_3$ also contains the invariant of $R_1 \cap R_2$), we decide that $I_i$ does not affect the invariable part of the function's output and $I_i$ is therefore irrelevant to the traffic token.

Figure 4 explains how the technique works through a real-world example (a popular Chinese app `com.tjsinfo.mangguoVideo`). In function $a()$, $v4$ is related to a traffic sink, three inputs ($v1$, $v2$ and $v3$, all are strings) of $convertToHex()$ may have contribution to the invariable portion of $v4$. To establish the connection, a static analysis of $convertToHex()$ seems necessary. However, the connection between the inputs and the potential invariant within the return variable $v10$ (at Line 10) can be hard to analyze, due to the complexity of the function. Therefore, here we resort to the coarse slicing and differential analysis. More specifically, through the coarse slicing, we know that $v2$ has no relation to any source of invariant, while both $v1$ and $v3$ may come from constants. So we randomly choose two values for $v2$ ($abcd$ and $efgh$) when setting $v1$ to a fixed value $abcd$ and $v3$ to $abcd$, running the function, recording its return values $v10_1$ ($abcd794fd8df6686e85e0d8345670d2cd4ae$) and $v10_2$ ($abcd03ac593fa7146ea182eeb2eb44d4dcfa$) and identifying their common part $abcd$. Then we randomly choose another value $efgh$ for $v3$ and observe that the return value $v10_3$ ($abcd36e2d7c526fd876eb14cd0b3ea2a3d43$) has the same common part as the intersection between $v10_2$ and $v10_3$, which indicates that $v3$ has no relation with the intersection. However, once we change $v1$ and execute the function again, the common part of the variables changes as well. This demonstrates that the impact of $v1$ on the invariant portion of the function's output cannot be ignored. So Tiger has to continue slicing the program with regard to that variable $v1$. In this way, our approach avoids working on the variables clearly unrelated to the invariant target, thereby significantly reducing the analysis time (see Section 4.3).

## 3.3 Optimization and Imprint Generation

Through the coarse slicing and differential analysis, Tiger can already significantly reduce the size of the slice for a network sink. Here we show that the slicing process can be further simplified by reusing the findings about the methods analyzed before. The slice generated in this way is further executed to discover traffic tokens and constructing app imprints, which also is elaborated below.

**Cross-slice optimization**. When running the IPE engine over an app, we can expect that a number of functions will be invoked again and again. This presents an opportunity for further optimizing the slicing process and simplifying the slice produced. More specifically, Tiger was designed to avoid repeated analysis of the same code and leverage existing PDT whenever possible. When slicing with regard to a network sink, the IPE creates a profile for each function it evaluated, which records whether the function contains invariant sources and which inputs affect the invariant on the function's output. These profiles are used when the IPE moves onto other network sinks within the same app: for all functions already profiled, their input and output variables are either immediately added onto the sink's PDT (when the variables are relevant) or instantiated (when they are not). This treatment further reduces the workload of the slicing operation.

**Partial execution**. Once a slice is generated, it needs to be executed to produce traffic tokens for imprint generation. The slice is in the form of a tree, which is rooted at its network sink statement. To run the slice, the IPE engine extracts its individual paths, from each leaf to the root, and then loads them to a modified Dalvik virtual machine (VM) for execution. Also, during the differential analysis, we need to run a method over different input values to identify those irrelevant to the output invariants. This step also relies on the modified VM.

[4] Note that here we only consider the primitive types (int, float, boolean, char) and serializable objects (e.g., string, time). The value of a primitive variable is randomly chosen in its data range. For a serializable variable, we randomly choose a value for each of its field.

The standard Dalvik VM runs on the mobile platform and can only process well-formatted Android APKs. In our research, we ported Dalvik VM to the desktop platform for large-scale evaluation and leveraged its support of Java reflection [2] to run individual statements on a slice. A challenge here comes from Android framework APIs. A direct loading of all their code into the VM is too heavyweight and also error-prone: for example, the system may crash when the API asks for the support from hardware. Our solution is API modeling, a common approach used in binary code analysis [19, 43] that summarizes the operations an API performs on given inputs. In our research, we found that most APIs encountered during the backward slicing do not contribute to the invariants (e.g., those returning the current geo-location and time). Therefore, their outputs can also be instantiated: that is, assigning the variable receiving the output a random value according to its type. However, there are a set of APIs highly related to traffic invariants, particularly those returning package name and other content from the manifest and other resource files, such as *android.os.Bundle.getStringArray*() for reading from the manifest and *android.content.res.Resources.getString*() for accessing the resource files. In sum, five categories of APIs are modeled[5]. These APIs are application independent and their functionalities are performed outside the VM before the results are delivered back to VM while calling these APIs in a slice.

**Imprint generation**. To discover invariable tokens and generate an app's imprints, Tiger runs each path on a slice twice with different parameters (different values for instantiating irrelevant variables). The common part of the traffic (a collection of common keys, values and other URL elements on an HTTP message) is then identified as the traffic token for the path. After identifying the invariants for all slices within an app, Tiger compares these token collections (one for each slice) among them to remove duplicates.

Although these token collections are invariable for an app, they may not be qualified as an app's imprints. This is because some tokens can be very general, shared across multiple apps. An example is shared libraries producing traffic invariants. These tokens or token collections show up in the communication of any app integrating the libraries and therefore cannot be used as an app's imprint. To address this issue, Tiger performs a cross-app unique content check, which compares each app's token collections against those of others. A token is dropped once it is also found on a different app's traffic. By running this check across a large number of apps (over 200,000 in our research), we have a reason to believe that the invariants left are specific enough to represent their corresponding apps. These invariants are therefore reported as the app's imprints.

Imprints are then sent to NGFWs to detect PHA or provide access control. In the process of detection, whenever a NGFW finds a packet containing all tokens of an imprint that Tiger generates, we say the app is identified[6].

**Table 1: App collection**

| App Source | | Count |
|---|---|---|
| GooglePlay | | 25,750 |
| Third-Party Markets | 360 | 6,177 |
| | Xiaomi | 600 |
| | Huawei | 3,592 |
| | Wandoujia | 1,235 |
| | Anzhi | 3,891 |
| | AppChina | 848 |
| | ChinaTelecom | 2,290 |
| VirusTotal | | 159,481 |

## 4 EVALUATION

In our research, we put our implementation to test using 200,000 real-world apps. In particular, we evaluated the effectiveness of Tiger by measuring its capability to capture identifiable traffic and false detection rate. Our study shows that Tiger is able to achieve extremely high coverage of capturing identifiable traffic, detecting >98% of identifiable packets from real-world app traffic, 43.98% more packets and 16.34% more apps than the prior approach [39]. The false detection rate is only 0.742%. In the meantime, our IPE technique turns out to be highly efficient: running against the standard techniques for slice generation and evaluation, our approach performs at least one order of magnitude faster.

### 4.1 Setting

Here we describe the apps collected in our study and the hardware and software settings for the experiments.

**App collection**. We crawled real-world apps from various sources last year, and got 203,864 apps after removing duplicated ones according to their MD5 checksums: 44,383 apps from Google Play and third-party Android markets covering every category provided by these markets (e.g. social, business, etc.); and 159,481 most recent PHAs (till June, 2016) collected from VirusTotal [4]. Everyday, nearly 800 thousands distinct samples are uploaded to VirusTotal for scanning, supporting the most up-to-date PHA samples covering wide range of malicious behaviors for studies [5]. The detailed information about these apps is presented in Table 1.

**Platform**. All the experiments were conducted on two servers running Ubuntu. One has 40 cores with 2.0GHz CPU, 256GB memory and 70TB hard drivers and the other has 20 cores with 2.1GHz CPU, 128GB memory and 30TB hard drivers.

### 4.2 Effectiveness

The most important for understanding the efficacy of our technique is the coverage it can achieve, in terms of the number of apps recognized from their traffic and the portion of the traffic attributed to their apps. Here we report our experimental study that evaluated these key properties of our technique. Also, we analyze the impact of dead code, which could lead to the imprints not showing up in any app's traffic.

**App coverage**. To measure the coverage, we installed Android apps on emulators (Android 4.4) and triggered their network behaviors to see how many of them can be captured by Tiger. In theory, it is very difficult, even impossible, to trigger all network

[5] Also, note that these APIs are implemented in the same way from version 1.6 to 7.0 (current version), which means we only need to implement their desktop version for once. In case their implementations are changed in the future version, revising the modeling is straightforward.

[6] The techniques of efficiently capturing apps using given imprints are out of the scope of this paper. Actually, these techniques are mature in current NGFW companies.

Table 2: App & Traffic Coverage.

| | | **Package Name<br>& Ad-ID** | **Package Name<br>Ad-ID & Host** | **Tiger** |
|---|---|---|---|---|
| App Identified | All | 59.72%<br>(2986/5000) | 69.82%<br>(3491/5000) | 76.06%<br>(3803/5000) |
| | Benign | 80.20%<br>(2005/2500) | 83.88%<br>(2097/2500) | 85.80%<br>(2145/2500) |
| | PHAs | 39.24%<br>(981/2500) | 55.76%<br>(1394/2500) | 66.32%<br>(1658/2500) |
| Packet Coverage | All | 28.87% | 62.71% | 72.85% |
| | Benign | 40.08% | 82.19% | 91.16% |
| | PHAs | 17.66% | 43.23% | 54.54% |

behaviors of an app. In order to cover as many network behaviors as possible, the traffic for testing was produced by a human-guided UI probing which is based on a state-of-the-art automatic UI exploration tool from NetworkProfiler [14] to execute Android apps for 5 minutes, as suggested by NetworkProfiler. The tool try to cover most of the network behaviors of apps: it first randomly operates on UIs of an app, records the paths it has gone through and then heuristically generates new paths to guide more UI explorations. And we manually moved the tool out of the UI state once it gets stuck in. Considering it is impossible to install and dynamically run all the collected apps, we randomly selected 2,500 apps from the legitimate markets and 2,500 PHAs from VirusTotal for test. All the traffic generated during the process was recorded and scanned using these apps' imprints from Tiger.

As we can see from the Table 2, Tiger produced the imprints that successfully identified 76.06% (3,803/5,000) of the apps. By comparison, the traffic signatures proposed by the prior approach [39], including package names and Ad-IDs, captured 59.72% (2,986/5,000) apps, which is 16.34% less than those captured by Tiger. From the 16.34% coverage increase, we found 10.10% (= 69.82% − 59.72%) were contributed by domain names. Note that, without Tiger's IPE, simply searching strings that look like hostnames and using them as imprints may not get the 10.10% increase since some hostnames are dynamically generated from the code. We also compare the app coverage between those from legitimate markets and PHAs from Virustotal. Interestingly, only 39.24% PHAs can be covered by the prior approach [39], while Tiger improves the rate to 66.32%. Looking into the improvement, we found that Tiger is capable of finding new types of invariants within traffic which helped detect additional apps. We provide the details about these new imprints in Section 5. We also note that, although the PHA coverage is increased while compared with previous approaches, it is still not as high as the coverage of legitimate apps. This is mainly due to the shared tokens among malware (e.g., potential harmful libraries [11]). Since our goal is to distinguish each unique app from other, we did not catch them in current implementation of Tiger. However, it would not be difficult to enhance our approach to identify the PHAs. For example, a straightforward way is to keep the shared tokens between PHAs.

We are also curious of the reason why Tiger's imprint missed the detection of the rest 1,197 apps. To find such reason, we have to manually check the generated imprints and the corresponding code in the apps. Considering that it is very hard to analyze all the 1,197 apps, we randomly selected 100 apps (around 10%) from them. And we found that 90 apps produce traffic tokens from the shared libraries they integrate (which appears on the traffic of any apps utilizing these libraries) and/or downloads resources from generic domains like http://qzone.qq.cn. These traffic tokens are not unique for a single app, hence cannot be used as imprints for app identification. Excluding these 90 apps that do not have identifiable imprints, Tiger only missed 10 apps due to the limitation of IPE on processing loops (see Section 6). In other words, for the apps that can be fingerprinted (including 3,803 identified, and 120 $(= 1197 \times (10/100))$ possible missed by Tiger), our approach caught 96.94% $(= 3803/(3803 + 120))$ of them.

**Traffic coverage**. Further we studied the traffic coverage which is estimated using the percent of the packets that carry the imprints for uniquely identifying these individual apps. This "traffic coverage" ratio is important, as it is critical in determining *how timely and likely an app can be recognized from its network traffic.* The more packets emitted by an app can be fingerprinted, the sooner and more likely the app can be detected. Especially, when mobile users walk from one networks (e.g., 4G and Wi-Fi) to another one and some of their apps' identifiable packets could be missed by these networks. Still, considering the cost of dynamic analysis, we use the 5,000 randomly sampled apps as those in app coverage evaluation.

In our experiment, we found 72.85% packets of the apps sampled were detected using the imprints created by Tiger, while only 28.87% could be identified by those generated by other approach [39], as shown in Table 2. Again, this difference (43.98%) is caused by other types of invariants discovered by Tiger through partially executing these apps' sink related code. As a result, the apps become easier to detect and more likely to identify using our new technique. In order to understand the imprints' coverage of truly identifiable packets, we further randomly selected 100 apps and manually inspected all 3,872 packets in their communication. Among them, 2,844 carried identifiable imprints, with 2,803 packages caught by the imprints from Tiger. Altogether, we conclude that Tiger is capable of capturing 98.56% $(= 2,803/2,844)$ of the packets carrying identifiable invariants. This level of coverage even exceeds what could be attained with a *perfect* training set: assuming that all URL-related invariants can be recovered from the perfect training set using other prior approaches [14] which is hard in practice; still the coverage that could possibly be achieved using package names, Ad-IDs and also the learnt hostnames was found to be no more than 62.71%, which is 10.14% below our approach, due to the new invariable tokens discovered by Tiger never reported before (see Section 5.2). Further, by looking into the 41 $(= 2,844 - 2,803)$ packets our approach missed, we found that in all these cases, the IPE engine recovered most part of their imprints. What it did not do right include one or two missing tokens caused by inaccurately processing a loop (which our approach only unwinds one iteration) that brings in erroneous invariants.

**False detection**. We also want to see how many apps were incorrectly captured by the imprints generated by Tiger. Since this evaluation does not need to dynamically run the apps, we used all of the 200,000 apps that we collected in this evaluation. In detail, we randomly selected 50,000 apps from 200,000 apps and generated their imprints. Then we checked the false detection using the rest 150,000 apps. That is, for every app in the rest 150,000 apps, we checked whether it can be identified by the imprints from 50,000 apps. If the identified app is not the one that generates the imprint, we view it as a false detection. Here, to judge whether two apps

Table 3: False detection

| | # Total Apps | #(%) False Detection |
|---|---|---|
| GooglePlay | 20,028 | 31 (0.155%) |
| 360 | 4,651 | 17 (0.366%) |
| Xiaomi | 450 | 2 (0.444%) |
| Huawei | 2,702 | 20 (0.740%) |
| Wandoujia | 936 | 8 (0.856%) |
| Anzhi | 2,927 | 11 (0.376%) |
| AppChina | 637 | 8 (1.256%) |
| ChinaTelecom | 1,731 | 29 (1.675%) |
| VirusTotal | 122,953 | 1,008 (0.820%) |
| *All Apps* | *152,897* | *1,134 (0.742%)* |

Table 4: Performance of IPE.

| | Full Slice | Coarse Slice | Tiger |
|---|---|---|---|
| **Total Time Cost** | 314 hours | 134 hours | 25 hours |
| **Avg Time Cost** | 226.380 s | 96.482 s | 18.227 s |
| **# of Total Node** | 155,839,237 | 101,082,209 | 9,214,422 |
| **# of Avg Node** | 31,168 | 20,216 | 1,843 |

are the same, we performed a strict rule: if the two apps have different MD5 values, we view them as different apps. Considering that we have removed the redundant apps with the same MD5 in our dataset, any identified app from the 150,000 apps should be a false detection. Based on this strict rule, we found that the false detection rate is only 0.742%. We also measured the false detection rate for each market (in Table 3). For the apps from Google Play, the false detection rate is very low (only 0.155%). We also found the apps from some Chinese markets have high false detection rate. This may be due to the repackaged apps, which are mis-identified by the imprints of the original apps.

**Impact of dead code**. Some imprints from Tiger may not match any app's traffic, since their corresponding network sinks may sit in a chunk of dead code. In our research, we estimated the ratio of such non-functioning imprints by running a reachability test on the sinks of the randomly selected 5,000 apps. Among the 9,488 network sinks discovered from these apps, 2,005 (21.13%) of them could not be reached from any entry point. Note that imprints produced by our approach are supposed to be used in NGFW, which is optimized to filter traffic with a large number of signatures. Redundant signatures of a moderate scale (such as 21.13%) only have a marginal impact on the firewall's performance. Therefore, we did not integrate into our implementation any mechanism for dead code removal.

## 4.3 Performance

Tiger is designed for discovering app traffic imprints on a massive scale. Key to this mission is high performance, which our unique design, the IPE in particular, strives to achieve. Here we report our effort to understand whether the new technique is indeed up to this high performance expectation, compared with its more straightforward alternative [32]. This evaluation was conducted on the randomly selected 5,000 apps from our dataset.

**Benefit of IPE**. We ran Tiger against the conventional slicing and execution approach, as proposed by the prior research [32]. In the experiment, we set the timeouts in both approaches to 10 minutes. The results of this study are presented in Table 4. As we can see here, the total time the conventional approach, which requires a full slicing, took to process all these apps is 314 hours on one of server with 20 cores. In contrast, our IPE technique spent merely 25 hours going through all these apps. On average, our approach needed 18 seconds per app, while the conventional counterpart took 226 seconds[7]. Table 4 further shows the number of pruned nodes from full slices. As we can see, on average, 94.09% $(= (31,168 - 1,843)/31,168)$ of nodes on a slice were pruned by the IPE, which resulted in a significant enhancement of the performance. Overall, Tiger achieved 12.42× $(= 226.38s/18.227s)$ speed-up compared with the conventional approach.

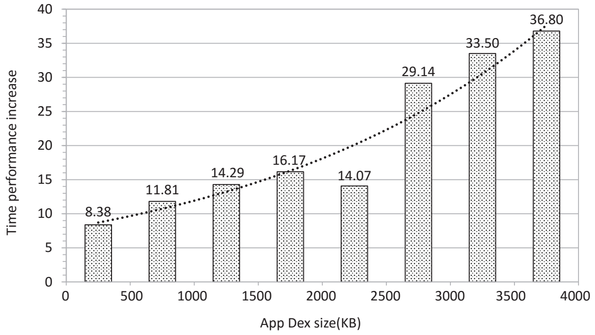


**Figure 5: Performance increase between instantiated slicing and full slicing regarding to app size.**

**App sizes**. We also compare the performance increase between instantiated slicing and full slicing regarding to app sizes (Figure 5). From the figure, we found the performance of Tiger increases more when an app has a larger size.

# 5 MEASUREMENT

By analyzing the traffic tokens derived from the code of over 200,000 real-world applications, a scale never attained before for this type of research, we were able to gain an unprecedented understanding of these imprints and their connections with today's Android apps, including their uniqueness and effectiveness in app identification, their relations with an app's functionalities and the conditions for triggering their related network traffic. Particularly, in addition to known invariants, we discovered other unexpected types of content that uniquely characterize a large number of popular apps. Further, from the content of some invariants, we could even figure out an app's operation environments (e.g., the permissions possessed by a library's host app). Our study also shows that the traffic involving some highly identifiable imprints cannot be easily triggered by automatic exploration tools like monkeyrunner, though the related functionalities can probably be invoked by human users frequently.

## 5.1 Landscape

**Imprint generation**. From the 203,864 apps, 181,582 apps have network sinks, and among them, Tiger discovered 392,645 imprints in total, as summarized in Table 5. Except the PHAs from VirusTotal, the apps from other sources, including Google Play and other third-party marketplaces, all have very high identification rates:

[7] The time is different from that reported in the prior work (150 seconds) [32]. This is mainly because the sinks that Tiger cares (HTTP-related APIs) is much more than that of HARVESTER (SMS-related APIs). Also, to be fair, we did not let HARVESTER use Soot to slice, which is known to be slow. We re-implement the slicing algorithm using the same technique as our IPE.

**Table 5: Apps and their imprints**

(Note that most unidentified apps are repackaged ones, which do not have unique imprints and are not identifiable from their traffic.)

| | # Total Apps | # (%) Apps Having Traffic | # (%) Identifiable Apps | # Imprints | # Imprints per App |
|---|---|---|---|---|---|
| GooglePlay | 25,750 | 25,183 (97.80%) | 24,753 (98.29%) | 89,678 | 3.62 |
| 360 | 6,177 | 5,986 (96.91%) | 5,830 (97.39%) | 40,316 | 6.92 |
| Xiaomi | 600 | 585 (97.50%) | 571 (97.61%) | 4,590 | 8.04 |
| Huawei | 3,592 | 3,483 (96.97%) | 3,423 (98.28%) | 25,126 | 7.34 |
| Wandoujia | 1,235 | 1,160 (93.93%) | 1,113 (95.95%) | 8,996 | 8.08 |
| Anzhi | 3,891 | 3,829 (98.41%) | 3,751 (97.96%) | 13,496 | 3.60 |
| AppChina | 848 | 821 (96.82%) | 769 (93.67%) | 4,878 | 6.34 |
| ChinaTelecom | 2,290 | 2,122 (92.66%) | 2,007 (94.58%) | 14,665 | 7.31 |
| VirusTotal | 159,481 | 138,413 (86.79%) | 97,748 (70.62%) | 202,393 | 2.07 |
| *All Apps* | *203,864* | *181,582 (89.07%)* | *139,965 (77.08%)* | *392,645* | *2.81* |

above 93%, with more than 98% of Google Play apps being identified by their imprints. In the meantime, PHAs from VirusTotal have lower identification ratio (70.62%). To find the reason, we checked the PHAs and found that a portion of PHAs from VirusTotal cannot be uniquely fingerprinted, due to their network sinks all present in shared code, including shared libraries, code templates or other apps they repackaged [9]. The invariants extracted from these apps also appear on the traffic produced by other apps using the same shared code. Although, we discovered that from all 40,665 such apps, only 920 unique certificates were recovered; also the app pairs we randomly sampled are almost identical in their code, except some differences in their resource files unrelated to network activities. As a result, all the invariants discovered from these apps are actually shared by their individual families. In our research, we discovered that these PHAs within some families can be unambiguously identified. Altogether, 77.08% of these 181,582 were found to be identifiable. On average each app has 2.81 imprints. Table 5 summarizes the identifiability of the apps from different sources. In each category, we can see the average number of imprints per app. The apps from VirusTotal have the least.

**Invariable token**. As mentioned earlier (Section 4.2), the imprints discovered by Tiger cover more packets than the traffic signatures produced by prior approaches [14, 39]. Fundamentally, the advantage comes from new types of invariable traffic tokens recovered from apps (Section 5.2). Table 6 presents the types of tokens we found, including not only domain, IP and keys that are also used in the prior research [24] which generate signatures from a given traffic training set, but also time values, device information values and credentials that have never been reported before. Even for the categories with known invariable tokens, such as ID-Value, not only does it include package name, but it also contains other surprising identifiable information like hardcoded session ID values (Section 5.2). Further, even for the same keys like `appkey`, they could appear at different locations in an HTTP packet, like on a URL, or within the HTTP header or content. The latter has never been used to fingerprint an app, up to our knowledge.

Each imprint contains one or more tokens (a keyword or a value), whose length ranges from 9 to 338 bytes. For example, the imprint of an real app involves five tokens "`webservice`", "`command`", "`getimage`", "`session`" and "`itemid`". Even with a high coverage, individual tokens tend to be too generic for app identification. As an example, in our dataset, "`command`" is in the traffic of 133 apps and "`session`" relates to 214 different apps. Combinations of multiple ones are much more specific. In our research, we found that except the long tokens like package IDs, most imprints (53.47%) contain more than one invariable tokens.

**Table 6: Invariable token**

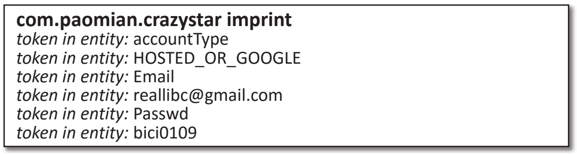
| | Category | Example | % |
|---|---|---|---|
| Key | ID/Key | appKey, appID, channelID, uid, sid, ... | 49.66% |
| | Time | time, date, timestamp,updateTime, ... | 6.92% |
| | Version | appVersion, apiVersion, osVersion, ... | 11.91% |
| | Device | imei, imsi, screen-width, screen-height, ... | 23.73% |
| | Http | Content-Type, Accept-Encoding, ... | 27.55% |
| | Unknown | hufplg, czkpln, lg, usd, ... | 30.24% |
| Value | Domain-IP | http://www.fotgtechnologies.com/, ... | 91.78% |
| | ID/Key | 555f89c2-94e6-8e8f-ffcb4d55a0c4, ... | 9.92% |
| | Time | 1936-07-09 06:34:05, ... | 0.39% |
| | Device | 3223423(imei), 23423(imsi), ... | 1.43% |
| | Credential | 280391LORE(password), ... | 0.15% |
| | Unknown | 13868388931, updateOpening, ... | 47.59% |

## 5.2 New Invariable Tokens

Among all the new tokens in Table 6, some are extremely intriguing. Here we take a close look at those most interesting ones, including fake device information, hardcoded time, login credentials and even the values of session IDs. The presence of these tokens also offers us a unique opportunity to better understand the backgrounds and operation environments of related apps only by imprints.

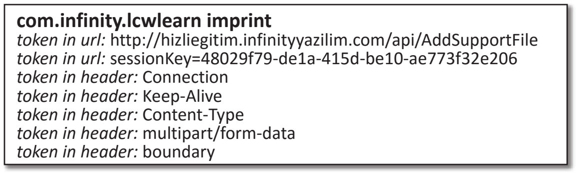
**Fake device information**. Since device information's values are supposed to change across devices and cannot be tied to a specific program, it has never been reported that an app can be identified by the *values* of these keys, such as "`imei`", "`imsi`'. Interestingly, we found in our dataset that such unique values do appear on the imprints of 2,826 apps. For example, an app sends out an URL: "`http://admin.ad-maker.info/...&device_id=94c24a0bc4fb8d342f0db892a5d39b4a`", regardless of the devices that it is running on. From its code, we found that the value "`94c24a0bc4fb8d342f0db892a5d39b4a`" is actually the md5 hash value of "`android_id`" and the related code statement is only executed when the app is running in background. Another example is that an app sends a fix value "a63123717ccbd3561a58808a6310e057" for device value ID when its host app does not have the `READ_PHONE_STATE` permission . Such fake device information can serve to fingerprint the target apps, and reveal some operation environments of the apps (e.g., being executed in background or having no `READ_PHONE_STATE` permission).

**Hardcoded time**. Time and date information often appears in network traffic. What is unexpected is that 768 apps in our dataset always send out the same time values in their traffic, which are actually embedded in their code. As an example, a bus app for users to choose suitable buses and communicate with each other adds "`date=2012-1-1 00:00:00`" to its HTTP message. After analyzing



**com.paomian.crazystar imprint**
*token in entity:* accountType
*token in entity:* HOSTED_OR_GOOGLE
*token in entity:* Email
*token in entity:* reallibc@gmail.com
*token in entity:* Passwd
*token in entity:* bici0109

**Figure 6: Example of Credential.**



**com.infinity.lcwlearn imprint**
*token in url:* http://hizliegitim.infinityyazilim.com/api/AddSupportFile
*token in url:* sessionKey=48029f79-de1a-415d-be10-ae773f32e206
*token in header:* Connection
*token in header:* Keep-Alive
*token in header:* Content-Type
*token in header:* multipart/form-data
*token in header:* boundary

**Figure 7: Example of Session ID.**

the app's code, we found that the HTTP request serves to retrieve from the app's server the chat logs always starting from this specific date. Also we found that 27 apps utilize their release time as a replacement for their version information. Such dates, again, are hardcoded and therefore can be used in imprints.

**Credential and personal data**. Prior research reports that many apps include hardcoded login credentials [21], which actually were found in the imprints generated by Tiger. For example, we found an app developer left an authentication key inside the app `dogantv.tv2`, when requests any resource stored in the developer's server. Since the data is hardcoded, the authentication key, therefore, becomes an invariable token for the app. Most interestingly, we saw some apps even include plaintext user names and passcodes within their code (see Figure 6), which were also observed in their traffic. These information was automatically extracted by our implementation to serve as a traffic token in the corresponding apps' imprints.

**Session ID**. Also surprising is our finding that even session IDs (or cookies) were hardcoded and picked up from their traffic by Tiger as their invariable tokens. As we know, a session ID is a piece of data that helps stateless network protocols identify a session. It is typically used to recognize a user after she logs into a website. Usually, session IDs should be generated dynamically. However, there are indeed apps using fixed session IDs, as presented Figure 7, which shows that an app sends the session key "`48029f79-de1a-415d-be10-ae773f32e206`" out to a server. A possible explanation here could be the need for convenient access to the server-side resource of an app without explicit login. Interestingly, some developers seem aware of the risk of disclosing this security-sensitive information to the public, and try to conceal the IDs in HTTP packet headers, instead of directly exposing them on URLs. For Tiger, however, this does not make any difference.

## 5.3 Triggers

By design, Tiger is well equipped to recover hidden identifiable tokens that tend to be missed by automatic exploration tools. This, however, does not necessarily mean that these tokens rarely show up in the app's traffic: on the contrary, such tokens could be good indicators for the apps when they are interacting with human users or performing operations of interest. Following we present the examples for the cases where network sinks become extremely hard to trigger automatically without looking into their code (which makes all existing learning-based approaches less effective).

**Human involvement**. For the automatic tool, a common example for a complicated trigger is an app's login page. We found that about 24% of apps hide their identifiable network behaviors behind login protection. Examples include Airbnb and iKuLing. Note that to enable an automatic tool to generate login traffic, one needs to manually enter a list of login credentials for different sites. For Tiger, however, this becomes unnecessary, as long as the code responsible for such behaviors is on the app side.

**Intent trigger**. Another case is an app whose identifiable traffic can only be invoked by an Intent issued by itself or other apps running on the same device. For example, the app "`com.pekingsjht.iyankerapp`" has a special "`SearchActivity`", which cannot be reached from its main view, and instead, can only be launched by an Intent. Clearly, Monkey and its variations will be almost impossible to trigger this activity. Actually, discovering the trigger condition is highly nontrivial, which may require a symbolic execution to recover the construction of the Intent. For Tiger, however, all we need is to find the related network sink and partially runs the slice to generate tokens, without constructing that Intent.

**Triggered from the remote**. Also discovered in our study is the network operations that are triggered by the content on a server. For example, there is an app includes a specific activity for video playing that can only be activated when a URL scheme like "aipai-vw://video/" displayed in the webview. The trouble is that the scheme may or may not show up behind certain widgets (such as a button) on the webpage within the webview and therefore becomes almost impossible for an automatic tool to trigger. As a result, the network operation in the video-playing activity simply cannot be invoked automatically. Again, such hidden activities are tricky for the automatic tool but can still be common when the app is being operated by a human user.

**Triggers of PHAs**. Triggers inside PHAs protect suspicious behaviors from dynamic analysis, which makes the concealed imprints difficult by UI exploration approach to catch. We found such triggers existing in our measurement. For example, we found a spyware "`com.nicky.lyyws.asl`", which is from the `NickyBot` family. Only when an SMS message with specific contents is received by the infected smartphone, the spyware is triggered to send the content of the received message to a remote server. Another case we found is "`com.wuzla.game.ScootrHero_Lite`", a malicious app from the `GoldDream` malware family. Its imprints can only be captured by activating the trigger through an `UploadMessages` command from the remote controller. Tiger captured these imprints successfully.

## 6 DISCUSSION

With its high performance and effectiveness demonstrated in our evaluation and measurement study, the current design and implementation of Tiger are still preliminary. Here we discuss a few limitations of the technique and potential ways to move it forward.

**Limitations of IPE**. Tiger could lead to the imprints that do not exist in any real apps' traffic, due to the limitations of the IPE technique and also the existence of dead code. The former is mainly caused by the challenges in processing loops (Section 4). One possible way to move forward is to explore the technique such as fuzzy matching: looking for the token combination close to, instead of exactly matching an imprint, given the observation that the errors within an imprint, when exist, tend to be minor, involving typically only one token. This may need to set up a threshold to judge whether two imprints are the same. Such threshold could be measured from future evaluations on the false detection and coverage rate. Another cause of the false negative is pruning of a variable related to an invariant output. Particularly, partial execution of a method can miss some of its internal paths, which could only be triggered by assigning some input variables with specific values, rather than the random concrete values given by our IPE engine. In other words, some input variables could be dropped as irrelevant ones but may actually affect the output invariants when they take certain specific values. Although theoretically feasible, we have not found any instance of this type from the randomly selected 100 apps after manual checking. What we observed actually are relatively straightforward, serving the purpose such as encoding(e.g., base64, MD5) and building packet content (e.g., storing payloads into a Hashmap). Also, Tiger achieves over 98% coverage using this approach. Another issue is dead code, which leads to the analysis on the sinks that will not be executed during an app's real-world operations. Our preliminary study shows that the impact of the dead code can be limited (Section 4). More effort is needed to understand the performance implications of dead code, which make a firewall screen traffic with more signatures than needed. In order to further reduce the impact of dead code, an efficient reachability test is planed to be built into our system.

**Imprint construction**. Further, the main objective of Tiger is to efficiently generate high-quality traffic. The technique for imprint construction is not the focus here and therefore may not be perfect. More specifically, we now just look at the combination of invariable tokens (e.g., key-value pairs) on the same flow, and remove duplicated tokens across apps. The imprint generated in this way does not capture some packets or even apps. For example, a repackaged app may not have any flow-based imprint, as its traffic invariants either belong to the app it clones or the advertising library injected into its code, though it can still be fingerprinted by the combination of these invariants. Imprint generation over multiple flows is challenging, due to the difficulty in linking these flows together (e.g., a TCP connection for retrieving data from the server and another connection for downloading ads). Prior research utilizes observed temporal relations to correlate two flows [28], which may not work in the case of a repackage app, when its identifiable flows are produced at different times. How to address this problem is left for the future research.

## 7 RELATED WORK

**Imprint generation**. Given the emerging demands for managing network behaviors of mobile apps, techniques for app traffic fingerprinting have been intensively investigated in recent years, both by the industry (such as Palo Alto networks) and the academia [6, 7, 12, 14, 24, 26, 28–30, 37, 42]. All existing techniques are based upon direct analysis of app traffic for imprint generation, mainly relying on supervised learning to build a classification model from a training set of network traces [30]. Most of these approaches assume that the app network traces are already given, sometimes from ISPs and mobile carriers [28]. In the other cases, automatic UI exploration techniques (e.g., monkeyrunner [1]) are directly used or enhanced to generate app traffic. For example, NetworkProfiler [14] improves monkeyrunner by leveraging recorded user events and other heuristics to discover new UI paths of the app under the test. Still little is known how comprehensive the traces produced in that way could be, in comparison with the app's real-world traffic, even after a substantial amount of time is spent on the testing (at least 5 minutes). In general, an app's realistic, comprehensive network traces are hard to come by, not to mention the challenge in generating such traces at a large scale, for millions of apps. Tiger is designed to address this challenge, which uses code-analysis to guide the traffic triggering and imprint generation. The only prior approach not relying on the training set or seed signatures has been proposed in a study on in-app advertisement [39]. The approach uses app names (i.e., package names) and AD-IDs collected from an app's meta data to fingerprint its traffic. Also, the effectiveness of package names has been mentioned in other work [24]. As demonstrated in our experiment, the invariants Tiger recovers from an app's code vastly outperforms these tokens in terms of their coverage on the app's traffic. Our findings show that indeed efficient code-analysis is a way to go for the large-scale app imprint generation.

**Program slicing**. Program slicing is a technique for simplifying a program by focusing only on a subset of its code relevant to some points of interest (aka sinks) [38]. It has been widely utilized in debugging/testing [22, 41], program behavior analysis [23, 27] and bug detection [25]. Due to the complexity of modern applications, scalability is always an issue for the real-world use of the techniques [34]. As a result, pruning is used to serve different purposes: for example, "thin slicing" [36] has been proposed to identify statements that produce incorrect values [44]. The slicing technique has also been applied to analyze Android apps. As an example, SAAF [20] utilizes slicing to backtrack the parameters of a given method. Harvester [32] combines slicing with dynamic execution for extracting runtime values of plain-text telephone numbers in SMS trojans, command and control messages of bots in malware. However, never before has the slicing technique been tuned towards the analysis of traffic invariants, using instantiation of unrelated variables to prunes the slice tree. Our research shows that this new technique achieves 12.42× speedup over the conventional slicing and execution, and may have the potential to be applied to other domains.

## 8 CONCLUSIONS

In this paper, we present Tiger, a novel technique that makes massive scale, comprehensive app imprint generation possible. At the center of Tiger is a unique instantiated partial execution technique that slices the code related to an app's network sinks in a way that the variables unessential to the app's network tokens are quickly identified and instantiated, and related paths are quickly pruned. As a result, only a very compact set of statements need to be run to recover the content of the invariable tokens, which enables at least one order of magnitude faster than a more conventional slicing and execution approach. In the end, Tiger is shown to achieve the performance of processing each app in 18 seconds on average and the effectiveness of covering over 98% of identifiable app traffic.

Running Tiger over 200,000 real-world apps, a scale never done before in traffic signature generation, we were able to gain an in-depth understanding about the identifiable traffic produced by modern apps. We found that unexpected information from apps' communication, including fake device information, hardcoded time, credentials and session IDs, could all be used to uniquely fingerprint individual apps. Also discovered in our work is the presence of complicated triggering conditions, requiring human intervention, Intent trigger and remote instructions, which demonstrates the limitations of automatic UI exploration and the importance for code-analysis based solutions. Finally we show the potential directions to move forward, particularly possible enhancement of our techniques to generate more complicated cross-flows imprints, and further improvement on the coverage of the packets that can be identified.

## ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive comments. And we also thank VirusTotal for the help in validating suspicious apps in our study. IIE authors were supported in part by NSFC U1536106 and 61728209, National Key Research and Development Program of China (Grant No.2016QY04W0805, No.2016YFB0801603), Youth Innovation Promotion Association CAS, strategic priority research program of CAS (XDA06010701). IU authors were supported by NSF CNS-1223477, 1223495, 1527141, 1618493, ARO W911NF1610127 and Samsung Gift fund.

## REFERENCES

[1] 2017. monkeyRunner. https://developer.android.com/studio/test/monkeyrunner/index.html. (2017).

[2] 2017. Trail: The Reflection API. https://docs.oracle.com/javase/tutorial/reflect/. (2017).

[3] 2017. UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html. (2017).

[4] 2017. VirusTotal. https://www.virustotal.com. (2017).

[5] 2017. VirusTotal file statistics during last 7 days. https://www.virustotal.com/en/statistics/. (2017).

[6] AddictiveTips. 2017. Easily Monitor All Incoming & Outgoing Network Connections On Android. http://www.addictivetips.com/android/monitor-all-incoming-outgoing-network-connections-on-android/. (2017).

[7] Hasan Faik Alan and Jasleen Kaur. [n. d.]. Can Android Applications Be Identified Using Only TCP/IP Headers of Their Launch Time Traffic. In Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec' 16), r (Ed.).

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014. 29.

[9] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In Proceedings of the 36th International Conference on Software Engineering. ACM, 175–186.

[10] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale.. In USENIX Security Symposium. 659–674.

[11] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. 2016. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In Security and Privacy (SP), 2016 IEEE Symposium on. IEEE, 357–376.

[12] Mauro Conti, Luigi V. Mancini, Riccardo Spolaor, and Nino Vincenzo Verde. 2015. Can't You Hear Me Knocking: Identification of User Actions on Android Apps via Traffic Analysis. In Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY' 15). 297–304.

[13] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. 2017. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS). 1–16.

[14] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. 2013. NetworkProfiler: Towards automatic fingerprinting of Android apps. In Proceedings of the 32nd IEEE International Conference on Computer Communications (INFOCOM' 13). 809–817.

[15] Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. 2008. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. IEEE, 161–166.

[16] Gartner. 2017. Managed Security Service Provider (MSSP). http://www.gartner.com/it-glossary/mssp-managed-security-service-provider/. (2017).

[17] Arnab Ghosh, Prashant Kumar Gajar, and Shashikant Rai. 2013. Bring your own device (BYOD): Security risks and mitigating strategies. Journal of Global Research in Computer Science 4, 4 (2013), 62–70.

[18] Google. 2017. The Google Android Security Team's Classifications for Potentially Harmful Applications. https://static.googleusercontent.com/media/source.android.com/en//security/reports/Google_Android_Security_PHA_classifications.pdf. (2017).

[19] Michael I Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Information-flow analysis of Android applications in DroidSafe. In Proc. of the Network and Distributed System Security Symposium (NDSS). The Internet Society.

[20] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. [n. d.]. Slicing droids: program slicing for smali code. In Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC' 13). 1844–1851.

[21] Anurag Kumar Jain and Devendra Shanbhag. 2012. Addressing Security and Privacy Risks in Mobile Applications. IT Professional 14, 5 (2012), 28–33.

[22] Mariam Kamkar, Peter Fritzson, and Nahid Shahmehri. 1993. Interprocedural Dynamic Slicing Applied to Interprocedural Data How Testing. In Proceedings of the Conference on Software Maintenance (ICSM' 93). 386–395.

[23] Bogdan Korel and Juergen Rilling. 1998. Program Slicing in Understanding of Large Programs. In Proceedings of the 6th International Workshop on Program Comprehension (IWPC' 89). 145–152.

[24] Anh Le, Janus Varmarken, Simon Langhoff, Anastasia Shuba, Minas Gjoka, and Athina Markopoulou. 2015. AntMonitor: A System for Monitoring from Mobile Devices. In Proceedings of the 2015 ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big (Internet) Data,. 15–20.

[25] Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. 2016. AntMiner: mining more bugs by reducing noise interference. In Proceedings of the 38th International Conference on Software Engineering (ICSE 2016). 333–344.

[26] Envato Pty Ltd. 2017. Analyzing Android Network Traffic. http://code.tutsplus.com/tutorials/analyzing-android-network-traffic–mobile-10663. (2017).

[27] Andrea De Lucia, Anna Rita Fasolino, and Malcolm Munro. 1996. Understanding Function Behaviors through Program Slicing. In Proceedings of the 4th International Workshop on Program Comprehension (WPC' 96). 9–10.

[28] Stanislav Miskovic, Gene Moo Lee, Yong Liao, and Mario Baldi. 2015. AppPrint: Automatic Fingerprinting of Mobile Applications in Network Traffic. In Proceedings of the 16th International Conference on Passive and Active Measurement (PAM' 15). 57–69.

[29] Sophon Mongkolluksamee, Vasaka Visoottiviseth, and Kensuke Fukuda. 2015. Enhancing the Performance of Mobile Traffic Identification with Communication Patterns. In Proceedings of the 39th IEEE Annual Computer Software and Applications Conference (COMPSAC' 2015). 336–345.

[30] Fairuz Amalina Narudin, Ali Feizollah, Nor Badrul Anuar, and Abdullah Gani. 2016. Evaluation of machine learning classifiers for mobile malware detection. Soft Comput. 20, 1 (2016), 343–357.

[31] Palo Alto Networks. 2017. WildFire Analysis Categories. https://www.paloalto-networks.com/documentation/autofocus/autofocus_admin_guide/assess-autofocus-artifacts/wildfire-analysis-categories.html. (2017).

[32] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques. In Proceedings of the Network and Distributed System Security Symposium (NDSS' 16).

[33] RFC. 2000. HTTP Over TLS. https://tools.ietf.org/html/rfc2818. (2000).

[34] Juergen Rilling and Tuomas Klemola. 2003. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In Proceedings of the 11th IEEE International Workshop on Program Comprehension. IEEE, 115–124.

[35] Smali. 2013. An assembler/disassembler for Android's dex format. http://code.google.com/p/smali/. (2013).

[36] Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. 2007. Thin slicing. In Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI' 07). 112–122.

[37] Jianhua Sun, Lingjun She andHao Chen, Wenyong Zhong, Cheng Chang, Zhiwen Chen, Wentao Li, and Shuna Yao. 2015. Automatically identifying apps in mobile traffic. Concurrency and Computation: Practice and Experience (2015).

[38] Frank Tip. 1995. A survey of program slicing techniques. Journal of Program Language 3, 3 (1995).

[39] Alok Tongaonkar, Shuaifu Dai, Antonio Nucci, and Dawn Song. 2013. Understanding Mobile App Usage Patterns Using In-App Advertisements. In Proceedings of the 14th International Conference on Passive and Active Measurement (PAM' 13). 63–72.

[40] Aliaksei Tsitovich, Natasha Sharygina, Christoph M Wintersteiger, and Daniel Kroening. 2011. Loop summarization and termination analysis. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 81–95.

[41] Mark Weiser. 1982. Programmers Use Slices When Debugging. Commun. ACM 25, 7 (1982), 446–452.

[42] Qiang Xu, Thomas Andrews, Yong Liao, Stanislav Miskovic, Zhuoqing Morley Mao, Mario Baldi, and Antonio Nucci. 2014. FLOWR: a self-learning system for classifying mobileapplication traffic. In Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'14). 569–570.

[43] Mu Zhang and Heng Yin. 2014. Efficient, Context-aware Privacy Leakage Confinement for Android Applications Without Firmware Modding. In Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (CCS' 14). ACM, 259–270.

[44] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Pruning dynamic slices with confidence. In Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI' 06). 169–180.