SARRE: Semantics-Aware Rule Recommendation and Enforcement for Event Paths on Android

Yongbo Li, *Student Member, IEEE*, Fan Yao, *Student Member, IEEE*, Tian Lan, *Member, IEEE*, and Guru Venkataramani, *Senior Member, IEEE*

***Abstract*— This paper presents a semantics-aware rule recommendation and enforcement (SARRE) system for taming information leakage on Android. SARRE leverages statistical analysis and a novel application of minimum path cover algorithm to identify system event paths from dynamic runtime monitoring. Then, an online recommendation system is developed to automatically assign a fine-grained security rule to each event path, capitalizing on both known security rules and application semantic information. The proposed SARRE system is prototyped on Android devices and evaluated using real-world malware samples and popular apps from Google Play spanning multiple categories. Our results show that SARRE achieves 93.8% precision and 96.4% recall in identifying the event paths, compared with tainting technique. Also, the average difference between rule recommendation and manual configuration is less than 5%, validating the effectiveness of the automatic rule recommendation. It is also demonstrated that by enforcing the recommended security rules through a camouflage engine, SARRE can effectively prevent information leakage and enable fine-grained protection over private data with very small performance overhead.**

***Index Terms*— Android privacy, rule recommendation, statistical analysis, path identification, rule enforcement.**

I. INTRODUCTION

With its increasing popularity, Android continues to claim the largest share of malware [1] among all smartphone platforms. It has been shown in recent research findings that 51.1% of Android malware collects and leaks users' private data such as accounts and SMS [2], and 45.3% of Android malware samples are able to send out background messages without triggering user awareness. In addition, users' private information can also leak out through apps downloaded from Google Play, as evidenced in [3]. Information leakage and privacy issue remains to be a challenging problem for hardening smartphone security.

The limitations of Android's current permission-based security mechanism have been well recognized in prior work, e.g., [2], [4]. A number of proposals are made to tackle this challenging problem by runtime security enhancement solutions based on Access Control [5], [6], static information flow analysis [7], and data obfuscation technique [8], [9]. However, the burden of manually constructing extensive security rules for various apps still lies with smartphone users or app developers, who may find it overly convoluted and difficult to adjust on the fly. The problem is further complicated when different information flows accessing the same data require differentiated security rules. For instance, while GPS coordinates are routinely queried by information flows in map/tracking apps, it could raise serious privacy concerns if they are accessed by an alarm clock app, whether it contains repackaged malware or benign ad libraries collecting user location data. Traditional automated software security specification methods based on static code analysis [10], [11] are not applicable to Android platform, because developers extensively adopt code encryption, code obfuscation and dynamic code loading techniques. Recent studies begin to investigate automated rule assignment in smartphone systems [12], but only consider a one-size-fits-all solution for each data source and fall short on providing fine-grained security rules for different information flows and app semantics.

Automated fine-grained semantics-aware rule enforcement is a long-standing hard problem. First, events that can be leveraged for malicious purposes co-exist with other events in a mobile platform, and analyzing all the runtime events together is not trivial. Second, the sequences of runtime events are further affected by user behaviors, OS scheduling of multi-processes and other runtime randomness such as network delays. Third, different types of sensitive data may be requested in an app and users may have varied security preferences towards different types of sensitive data. Besides, even the same data can be requested for different purposes, and under different contexts.

In this paper, we present SARRE, a Semantics-Aware Rule Recommendation and Enforcement system that enables automated security rule recommendation and enforcement to prevent information leakage. To tackle the first two challenges, we employ statistical inference to identify information flows through light-weight runtime event monitoring. In particular, SARRE constructs an event graph by correlation analysis between events. Based on that, the problem of identifying event paths is formulated as a minimum path cover problem in graph theory, which is NP-hard [13] and solved by our heuristic algorithm. The third challenge is solved by logging critical system events and semantic information

Manuscript received November 21, 2015; revised April 9, 2016 and July 1, 2016; accepted July 6, 2016. Date of publication July 28, 2016; date of current version October 11, 2016. This work was supported by the Office of Naval Research under Award N00014-15-1-2210. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Husrev T. Sencar.

The authors are with the Department of Electrical and Computer Engineering, The George Washington University, Washington, DC 20052 USA (e-mail: lib@gwu.edu; albertyao@gwu.edu; tlan@gwu.edu; guruv@gwu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2016.2596141

through our specially designed *Event Monitor*, associating information flows with the event paths identified and recommending security rules to event paths. To make a successful rule recommendation, SARRE employs collaborative filtering technique widely used in online recommendation systems [14] and leverages knowledge of (i) known security rules of similar paths and (ii) the corresponding app's semantic information. A recommendation is made for each new information flow by combining the rules of K nearest known flows with similar semantics. This approach allows SARRE to construct security rules that are both effective and in-context in an unsupervised manner. Finally, recommended security rules are implemented by a runtime enforcement mechanism that obfuscates data on information flows. Our entire SARRE system is prototyped on real Android devices and is evaluated with real-world malware samples and popular apps from Google Play, spanning multiple categories.

The main contributions of our paper are as follows:

1) We propose a novel approach to automate security rule construction for Android system. It makes novel use of collaborative filtering to build a recommendation system that assigns a security rule to each information flow based on both known path rules and application semantics, enabling fine-grained, semantics-aware protection of users' private data.

2) We develop a lightweight event monitor for runtime event logging and a path identification system to statistically infer information flows from event paths in real time. The problem is formulated as a minimum path cover in graph theory and a heuristic algorithm is developed to solve this NP-hard problem. Evaluating SARRE using 1706 app samples, including 1473 apps collected from Google Play and 233 malware samples, we show SARRE achieves 93.8% precision and 96.4% recall in identifying the event paths, compared with runtime tainting technique.

3) The proposed SARRE system is prototyped on Android devices. Using real-world malware samples and popular apps from Google Play, we show that the average difference between our automatic rule recommendation and manual configuration is less than 5%. Enforcement of the recommended rules demonstrates SARRE is effective to provide fine-grained protection over private data, while performance overhead remains small.

II. Motivational Examples

Two types of references utilized when making a rule recommendation in our system include: (i) similarity between event paths and (ii) semantic information. We use two sets of real-world examples to motivate their necessity, but our system is not limited to these two types of scenarios.

A. Location

Consider different uses of location data by three apps: (A) App *My Tracks* relies on accurate location data for tracking functionality. (B) Game *Drag Racing*'s main functionality doesn't rely on location data, but accesses it for advertisement purpose. Totally forbidding the access may make the app stop working, but user may want to limit the location data access to some limited accuracy level or with some noise included. (C) Malware *Nickispy* [16] repackages benign apps, but stealthily monitors GPS coordinates and sends the data out by network sockets.

To distinguish these scenarios, apps' declared functionality can serve as evidence to infer whether the runtime events are justified. Such evidence is obtainable by natural language processing used in prior work [17]. However, this is not enough because malware repackaging a game app shouldn't get the same accuracy level as a real game. To further distinguish this, runtime events presented as event paths can serve as another type of reference information. For advertisement purposes, games occasionally query the location by APIs such as *getLastKnownLocation*, especially when 'Activity'[1] is stopped or paused. In contrast, to track location stealthily, malware registers for *LocationListener* and retrieves location updates, used together with *getLastKnownLocation*, or *getLatitude, getLongitude*. By identifying the event paths, the above two cases can be further differentiated.

This example also justifies the need to monitor events that are not directly used for data access or transmission, but can serve as different triggers for event paths, and therefore help to differentiate paths and enforce semantics-aware rules. Such events include the critical system events like *GPS updated*, *SMS received*, *Activity's onStop method*, and etc.

B. Contacts Data

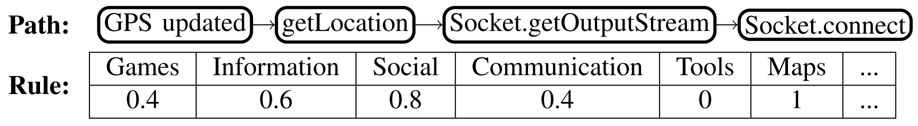
Consider different uses of contacts data by three apps: (A) Communication app *Truecaller*'s functionality relies on contacts data. (B) Social networking app *Facebook* only uses contacts data for purposes like inviting friends, and users may want to limit their access to certain limited availability level, for example, make only the email address and last name available. (C) Malware *Love Chat* [18] is disguised as a communication app, but harvests contacts data and sends it out to its server.

Apart from the reference information we discussed above in previous example, this example also motivates another type of semantic information, the parameters associated with API calls such as the destination address of a network socket. The malware *Love Chat* is disguised as a communication app, and exhibits similar event path, which queries contacts data through *ContentResolver* and sends it to remote server by *HttpPost*. By recording the parameters associated with API calls, the event paths can still be further distinguished and later associated with different rules.

III. Problem Description

An app's log file $\mathcal{L} = \{(e_1, t_1), \dots, (e_N, t_N)\}$ is a set of 2-tuples consisting of the event name e_n, and associated occurrence time t_n in the system. SARRE first employs statistical analysis to build an event graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ where each vertex represents an event and each arc represents a transition, and then identifies a set $\mathcal{P}$ of event paths on $\mathcal{G}$ for each app.

[1]'Activity' is a class name in Android app development to manage app's lifecycle.



Games	Information	Social	Communication	Tools	Maps	...
0.4	0.6	0.8	0.4	0	1	...

Fig. 1. An identified event path and its associated security rules for app *My Tracks* [15].

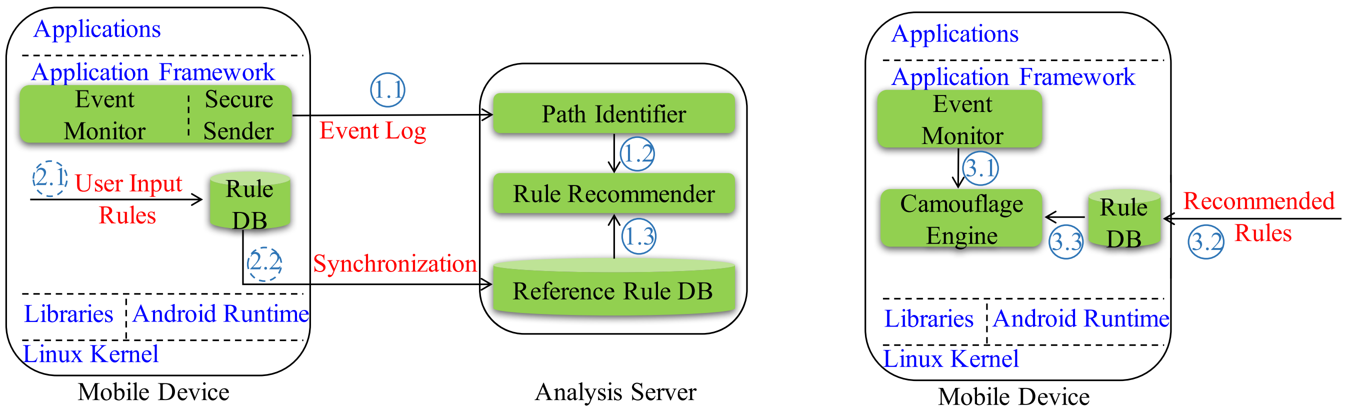


Fig. 2. Workflow diagrams of rule recommendation (left) and rule enforcement (right) phases. 1.1, 1.2 and 1.3 are steps for rule recommendation. 3.1, 3.2 and 3.3 are steps for rule enforcement. 2.1 and 2.2 are optional steps for users to overwrite recommended rules.

A security rule $R_r(0 \leq R_r \leq 1)$ is then to be assigned to each path $P_r \in \mathcal{P}$. The rule indicates the level of protection required. In particular, $R_r = 0$ implies that path P_r should be completely prevented from happening, while 1 leaves the path intact. Any other value corresponds to different level of protection and will be discussed in details in Section IV-D. Such security rule assignment needs to accommodate individual user's security preference. Besides, since similar paths can show up in different contexts and serve for different goals, the rule assignment needs to take semantic information into account.

SARRE makes use of collaborative filtering [14] technique to recommend a security rule for each path based on both known security rules of similar paths and semantic information. Declared functionality is one type of semantic information we consider. We use *DF* label to denote a label we assign to an app to represent its **declared functionality**. *DF* labels aim to represent apps' main functionalities and justify apps' needs for access to certain data or services. Obtaining apps' *DF* labels can be assisted by the category information provided by app markets and also natural language processing of app descriptions [17], [19].

An example of identified event path and some of its semantics-aware security rules for app *My Tracks* [15] are shown in Fig. 1. The path consists of four events, which access location data whenever GPS coordinates are updated, then transmit data via network socket. Such path is kept untouched ($R_r = 1$) for 'Maps' apps, while for 'Games' and 'Communication' apps, information flowing through this path needs different levels of security protection. In Section IV-D we will discuss how numerical rules map to different rule enforcement methods, depending on the types of data involved.

Threat Model: In this paper, our primary concern is the undesirable information leakage that exists in malware, as well as apps from Google Play. Information leakage flows can be associated with event paths in the system. All the applications are assumed to be untrusted on the mobile system, but we assume that the encrypted log files generated by our system won't be intercepted and decrypted by normal applications.

IV. Our SARRE System

This section provides our system design details and algorithms. We use two workflow diagrams to depict two key phases when rule recommendation (Fig. 2 left) and rule enforcement (Fig. 2 right) happen. In rule recommendation phase, *Secure Sender* sends the event logs generated by *Event Monitor* to *Path Identifier* 1.1, which will identify event paths from the logs and forward to *Rule Recommender* 1.2. Referring to *Reference Rule DB* 1.3, *Rule Recommender* then makes rule recommendations to event paths. Meanwhile, option is reserved for users to overwrite recommended rules in *Rule DB* on mobile devices 2.1, and the overwritten rules are immediately synchronized to *Reference Rule DB* 2.2. In rule enforcement phase, *Camouflage Engine* is made aware of the events by *Event Monitor* 3.1. Recommended rules are stored in *Rule DB* on mobile devices 3.2 and made available to *Camouflage Engine* 3.3, which will enforce the rules to monitored events. We dedicate each following subsection to one of the four main parts: (i) *Event Monitor*, (ii) *Path Identifier*, (iii) *Rule Recommender*, and (iv) *Camouflage Engine*.

A. Event Monitor

Event Monitor intercepts and logs the events (within a configurable list) at the framework level of Android system. Our scope of monitored events include: (i) apps' calls to

TABLE I
SUMMARY OF MONITORED APIS AND OTHER EVENTS

Resources and states	**APIs and other monitored events**	**Description**
Telephony related information and SMS	APIs exposed in *TelephonyManager*, *SmsManager*, e.g. *getCellLocation()*, *getLine1Number()*, *sendTextMessage()*, SMS broadcast receiver, SMS content observer.	SMS receiver listens to new SMS broadcasts, SMS content observer monitors changes of SMS.
Location based information	APIs in *LocationManager*, e.g. *getLastKnownLocation()*, GPS location listener, GPS broadcast receiver.	GPS location could be accessed by querying GPS directly or register for a GPS listener or receiver.
Accounts	APIs in *AccountManager*, e.g. *getAccountsByType()*.	
Application information	APIs in *ApplicationPackageManager*, e.g. *getApplicationInfo()*.	Information of installed applications can be retrieved to check whether Anti-virus or banking software is installed.
Resources provided by *ContentProvider*	query() API in *ContentResolver* with URI referring to contents like *call log*, *contact list*, and *browse history*.	*ContentResolver* provides access to content maintained by both system and applications.
File I/O and network I/O	APIs in *Preference*, *InputStream*, *OutputStream*, *Environment*, *AbstractHttpClient*, e.g. *getExternalStorage()*, *openFileOutput()*, *read()*, *write()*, *getInt()*, *putString()*, *execute()*.	Both sensitive information and malicious payloads can be kept on external or internal storage; they can also be transmitted through network I/O.
Android SQLite database	APIs such as *getWritableDatabase()*, *query()*, *queryWithFactory()*, *rawQueryWithFactory()*.	These APIs support accesses to other apps' content providers, and much other information stored in SQLite database.
Connectivity states information	APIs in *WifiManager* and *ConnectivityManager*, e.g. *getWifiState()*, *getNetworkInfo()*.	Changes of connectivity states are frequently used as triggers for undesirable behaviors.
Phone states information	Phone state content observer, broadcast receiver and listener, such as APIs in *PowerManager*.	Phone state changes can also serve as triggers for undesirable behaviors, e.g. screen on/off switch.
Apps' activity states	activity *onCreate()*, activity *onStop()* , service *onCreate()*.	Alike other states, apps' activity states can serve as triggers for undesirable behaviors.

standard APIs that can be leveraged for data collection, processing, and transmission, such as API calls to access location, query SMS, and network services; and (ii) other system events or phone state changes that are not directly related to information flows, but facilitate identification and characterization of them, for example, an incoming phone call, new SMS, and screen ON/OFF switch frequently serve as different triggers for information flows, with or without users' awareness. These system events or state changes help to build the full scenarios of event paths. For example, an event path in malware *Nickispy* [16] to access SMS and send it out by network sockets triggered by an incoming SMS can be distinguished from similar paths in data backup apps normally triggered by users' actions. Critical events related to sensitive information flows and frequently employed by malware have been widely studied in the literature [4], [20] and malware reports [21]. Based on these, we compose a list of 70 classes of key events for our prototype of SARRE's *Event Monitor*, as shown in Table I. In the table we summarize the sensitive resources and system states in the first column, and the corresponding APIs used to access the resources or listen to system state changes and system events in the second column. While this list shows sufficient events coverage in our study, it can be further expanded to provide more extensive information regarding various event paths and information flows.

There exists a tradeoff between the coverage of events and the performance overhead introduced by event monitor. A large list to monitor may unnecessarily cover non-critical events and cause high overhead, while a small list may neglect critical events, resulting in unsatisfactory path identification. To exploit this tradeoff, SARRE's rule recommender in Section IV-C formulates an optimization problem, which attaches a weight to each monitoring event and selects events that are critical for path identification and rule recommendation by optimizing the weights on a large training set. Basically, it starts with a large set of monitored APIs and later leaves out those APIs with small weights from monitoring list. The optimization decision is fed back into event monitor to customize it for the optimal coverage and performance tradeoff. Details about our optimization-based approach for selecting monitored events is discussed in Section IV-C.

Considering that the same event may be employed by different event paths serving different purposes, to accurately construct event paths, we must be able to differentiate the same event attributed to different execution points in the same app. To achieve this, apart from the names of events, we also record the package name, the line number where an event is activated, and key parameters (e.g. the recipient's address of *sendTextMessage* API), all of which are hashed together to uniquely identify an event and further differentiate various event paths associated with the same event. The sensitive data returned to API calls on event paths are not logged in the log files to protect users' sensitive data. The log files only contain a sequence of events occurred, the unique hashed value and associated timestamps information. The log files are encrypted and transmitted periodically to *Path Identifier* by *Secure Sender*.

B. Path Identifier

Path Identifier quantifies the transitions between events within the log file for each app, then uses our path cover algorithm to extract the event paths.

In association rule mining problems [22], the metric *confidence* of transition $e_i \Rightarrow e_j$ is measured by the number of transactions containing both events e_i and e_j, divided by that containing e_i. To identify event transitions, we extend

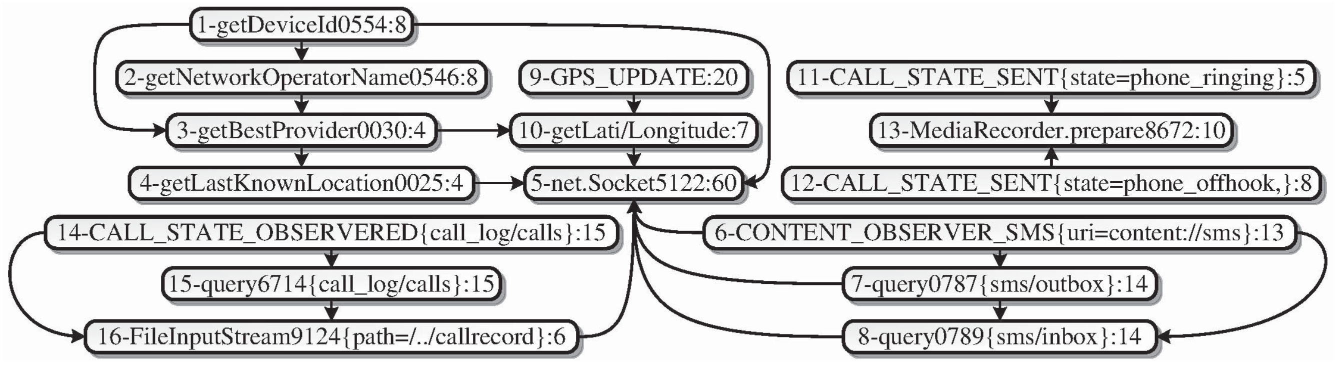

1-getDeviceId0554:8
2-getNetworkOperatorName0546:8
9-GPS_UPDATE:20
11-CALL_STATE_SENT{state=phone_ringing}:5
3-getBestProvider0030:4
10-getLati/Longitude:7
13-MediaRecorder.prepare8672:10
4-getLastKnownLocation0025:4
5-net.Socket5122:60
12-CALL_STATE_SENT{state=phone_offhook,}:8
14-CALL_STATE_OBSERVERED{call_log/calls}:15
6-CONTENT_OBSERVER_SMS{uri=content://sms}:13
15-query6714{call_log/calls}:15
7-query0787{sms/outbox}:14
16-FileInputStream9124{path=/../callrecord}:6
8-query0789{sms/inbox}:14

Fig. 3. Event Graph of the *Nickispy* malware. Each vertex is an event, uniquely identified by event name, key parameter, and last 4 digits of hashed line number. Each event's number of occurrence is shown in the end of vertex.

this metric to construct two novel measures: *forward confidence* μ_f and *backward confidence* μ_b, which respectively measure the confidence of two hypothesis testing "e_i is followed by e_j" and "e_j is preceded by e_i". Let $o(e_i)$ and $o(e_j)$ be the numbers of occurrences for events e_i and e_j in a log file. Further, we calculate the number of instances that event e_j happens after e_i and that event e_i happens before e_j, denoted by $o(e_i \rightarrow e_j)$ and $o(e_i \leftarrow e_j)$, respectively. This gives us the definition of forward and backward confidence (μ_f and μ_b) as follows:

$$\mu_f(e_i, e_j) = \frac{o(e_i \rightarrow e_j)}{o(e_i)}, \quad \mu_b(e_i, e_j) = \frac{o(e_i \leftarrow e_j)}{o(e_j)} \tag{1}$$

The definition of μ_b and μ_f is not redundant. These two metrics measure directional relationship between events. Consider an example app that frequently conducts network activities, at the same time, the app also occasionally queries sensitive information from the user devices, afterwards, it always opens a network socket to transmit it out. Assuming e_i denotes the events for sensitive data query, and e_j denotes the events related to network activities, such high correlation between the two types of events is captured by the significant forward confidence $\mu_f(e_i, e_j)$, while the backward confidence $\mu_b(e_i, e_j)$ is not significant. Similarly, the backward confidence μ_b captures the correlation in the opposite cases. In the special cases when e_i and e_j always happen together in pairs, the forward confidence and backward confidence are equal.

We say transition between e_i and e_j exists if the unified confidence $\mu_u(e_i, e_j)$, i.e. maximum of μ_f and μ_b exceeds threshold γ:

$$\mu_u(e_i, e_j) = max(\mu_f(e_i, e_j),\ \mu_b(e_i, e_j)) \geq \gamma, \tag{2}$$

which allows us to mine event transition between any pair of events in log files. It results in an event graph, which collectively aggregates all event transitions of a given app and is formally defined as Definition 1.

Definition 1: An **Event Graph** is a weighted, directed graph $\mathcal{G} = (\mathcal{V},\ \mathcal{A})$ with a set $\mathcal{V}$ of vertices and a set $\mathcal{A}$ of arcs, satisfying:

(i) Each vertex v_i in $\mathcal{V}$ represents a unique event in log file,
(ii) An arc (v_i, v_j) exists if and only if the unified confidence satisfies $\mu_u(v_i, v_j) \geq \gamma$. Each existing arc (v_i, v_j) is assigned $\mu_u(v_i, v_j)$ as non-negative weight.

The event graph constructed for a malware sample *Nickispy* (package name *com.nicky.lyyws.xmall* [16]) is depicted in Fig. 3. On the graph, each arc represents an observed transition, and is assigned with a weight equal to the *unified confidence* measured between arc's head and tail vertices by Equation (2).

Paths Identification on Event Graph: We formulate the Application Path Identification Problem as a variation of the minimum path cover problem [13], finding minimum number of paths $\mathcal{P} = \{P_1, \ldots, P_r, \ldots, P_R\}$ with a set of path counts $\{c_r,\ \forall r\}$ to cover the graph $\mathcal{G}$. Unique considerations exist in our problem from the traditional path cover problem: First, since a larger arc weight $\mu_u(v_i, v_j)$ implies higher confidence of the transition between two events e_i and e_j linked by the arc, optimal set of paths covering $\mathcal{G}$ should have the largest accumulated arc weights. Second, using $o(v_i)$ to denote number of occurrences of vertex v_i in the log file, to ensure identification of those paths with infrequent appearances in log files, a sufficient path cover of $\mathcal{G}$ should cover each vertex at least $o(v_i)$ times. This ensures all appearances of events during runtime are accounted for in path identification.

The standard path cover problem for general graphs is proven to be NP-hard [13]. We propose effective heuristics based on greedy algorithm to solve the Path Identification Problem. To be specific, our algorithm iteratively picks a source (i.e., the first vertex in a path) from graph $\mathcal{G}$ and grows a path by iteratively adding tail vertices until a sink vertex (vertex without children) is reached. In each step to add a tail vertex v_j, the algorithm picks the one with largest arc weight $\mu_u(v_i, v_j)$ among all candidate tail vertices. Once a path P_r is constructed, the algorithm calculates the minimum remaining number of occurrences of the vertices on the path, and assigns it to the path as the path count c_r. Then, c_r is deducted from remaining numbers of occurrences of the vertices. Due to the monotonicity of remaining number of occurrences, the algorithm is guaranteed to converge. The algorithm is summarized in Fig. 4. The paths identified for the malware sample *Nickispy* mentioned before, with event graph depicted in Fig. 3, are shown in Table II. The accuracy of path identification is validated in Section V-B.

TABLE II
LIST OF PATHS IDENTIFIED BY OUR PATH COVER ALGORITHM FOR SAMPLE *Nickispy*

Path	Explanation
$< 1, 2, 3, 4, 5 >$	Phone information and location leakage flow via network socket
$< 6, 7, 8, 5 >$	SMS leakage flow via network socket triggered by new SMS entry
$< 9, 10, 5 >$	Location leakage flow via network socket triggered by GPS updates
$< 11, 13 >$ and $< 12, 13 >$	Event paths to record phone calls with two different triggers (phone ringing and phone being off-hook)
<14,15,16,5>	Call log and recorded phone calls leakage flow via network socket triggered by newly-added call logs

```
Initialize $c(v_i) = o(v_i)$ for all $v_i \in \mathcal{V}$, $\mathcal{P} = \{\}$, r=0
// (a) Extract all paths from $\mathcal{G}$ until set $\mathcal{V}$ is empty
while $|\mathcal{V}| > 0$
  // (a.1) Build the next path
  Select source $v_i \in \mathcal{V}$ and initialize $P_r =< v_i >$, r++
  while $c(v_i) > 0$
    Find $x = \arg\max_{v_j} \mu_u(v_i, v_j)$, s.t., $(v_i, v_j) \in \mathcal{A}$
    Add vertex: $P_r \leftarrow< P_r, x >$, and update sink: $v_i \leftarrow x$
  end while
  Add path: $\mathcal{P} \leftarrow \mathcal{P} \cup P_r$
  // (a.2) Extract the path
  Set path count $c_r$ equal to the minimum remaining number
  of occurrences of the path's vertices
  for all $v_i \in P_r$
    Deduce $c_r$ from $c(v_i)$, remove $v_i$ from $\mathcal{G}$ if $c(v_i) == 0$
  end for
end while
// (b) Remove all paths' sub-paths
// (c) Merge fragmented paths
```

Fig. 4. Proposed algorithm to solve the Path Identification Problem.

C. Rule Recommender

1) Quantifying Path Similarity: Each identified path is an ordered sequence of events and transitions. Different events may have different levels of importance for identifying similar event paths and making rule recommendations. We introduce $\mathbf{W}$ as a set of non-negative weights and $w(e_i) \in \mathbf{W}$ as the weight assigned to event e_i. Such weights will be optimized later to minimize recommendation error. We introduce equation (3) to measure the similarity between two paths P_r and P_z:

$$s(P_r, P_z, \mathbf{W}) = \frac{\sum_{e_i \in (P_r \cap P_z)} w(e_i)}{\sum_{e_j \in (P_r \cup P_z)} w(e_j)} \tag{3}$$

We use '$\cap$' in (3) to denote *path intersection*, the longest common subsequence (event transitions) of paths P_r and P_z. Our notion of *path intersection* here has sequence meaning, because similarity of event paths not only depends on the common events, but also their transitions (i.e., ordering on the paths). For example, for $P_r = [1, 2, 5, 7]$ and $P_z = [2, 5, 1, 3, 7]$, $P_r \cap P_z$ should be $[2, 5, 7]$, rather than $[1, 2, 5, 7]$. To illustrate this, we consider the example in Fig. 1. If the last two vertices *Socket.getOutputStream* and *Socket.connect* happen before all other vertices, it becomes less likely that they are used to transmit sensitive data retrieved at other vertices of the path. Therefore, transitions between events are critical in defining path similarity. The '$\cup$' denotes *path union*, which is defined in this paper as the set of unique nodes of two paths.

2) Recommending Security Rules: Our objective is to recommend security rule $R_r \in [0, 1]$ to a path P_r whose parent app has given *DF* label, based on a set $\mathbb{P}$ of existing paths, their security rules $\mathbb{R}$, and their parent apps' *DF* labels. Our key idea is to use collaborative filtering [14], a technique widely used to build various recommendation systems. First, our recommendation algorithm searches for K-nearest neighbor paths denoted by $\mathbb{P}_K$, satisfying that (i) P_r and any $P_z \in \mathbb{P}_K$ share common *DF* labels, implying that the two paths' parent apps have matching semantics, and (ii) P_r and any $P_z \in \mathbb{P}_K$ have the highest similarity, i.e., $s(P_r, P_z, \mathbf{W}) \geq s(P_r, P_t, \mathbf{W})$ for all $P_t \in \mathbb{P} - \mathbb{P}_K$.

The set of K-nearest neighbor paths serves as reference to make semantics-aware security rule recommendation for path P_r. The recommended rule is calculated by a weighted average of the K-nearest neighbor paths' security rules:

$$R_r = \frac{\sum_{i:P_i \in \mathbb{P}_K} s(P_i, P_r, \mathbf{W}) \times R_i}{\sum_{i:P_i \in \mathbb{P}_K} s(P_i, P_r, \mathbf{W})} \tag{4}$$

where $s(P_i, P_r, \mathbf{W})$ is path similarity calculated by equation (3). The choice of weights $\mathbf{W}$ is important because some events such as sending a SMS are more critically related to sensitive paths than other events such as moving a cursor in SQLite database. Hence, these critical events warrant higher influence in calculating path similarity, which should not be biased by the number of events shown up in the paths. Besides, to avoid possible human bias introduced by manual weight assignment [23], we formulate an optimization problem to determine the optimal weights by minimizing recommendation error on training datasets.

3) Optimizing Event Weights: We divide the initial list of monitored events into 5 groups based on their functionality and relevance to sensitive resources [4], [20]. Using training dataset consisted of paths $\mathbb{P}_T$, with pre-configured security rule $\tilde{R}_r$ for each $P_r \in \mathbb{P}_T$, we calculate the recommended rule R_r using (4), and path similarity using (3). We minimize the overall square error between R_r and pre-configured rule $\tilde{R}_r$ by optimizing the weights $\mathbf{W}$, as shown in Equation (5).

$$\begin{aligned} &\text{minimize} \quad \sum\nolimits_{P_r \in \mathbb{P}_T} \epsilon(P_r), \\ &\text{subject to} \quad \epsilon(P_r) = |R_r - \tilde{R}_r|^2, \ 1 \succeq \mathbf{W} \succeq 0 \end{aligned} \tag{5}$$

Solving the optimization problem (5) serves as a training stage in SARRE's *Rule Recommender* to enable fine-tuning of the recommendation algorithm to fit each user's preference and security expectation. It also allows the recommendation algorithm to keep improving in an unsupervised fashion as

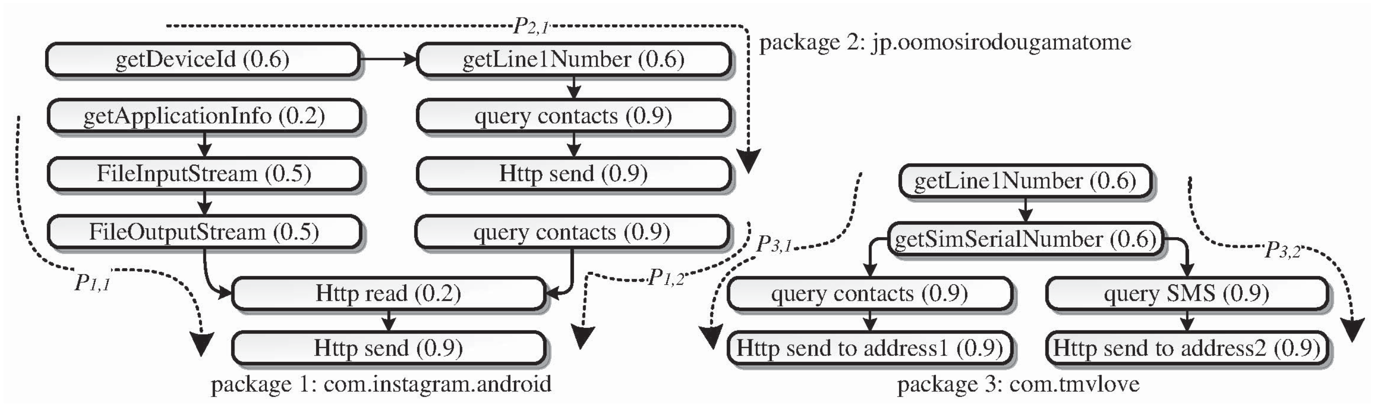


Fig. 5. Example consisted of 3 apps to illustrate the *Rule Recommender*. The end of text in each vertex is the event's weight.

more knowledge (i.e., security rules and paths) becomes available on the fly.

4) An Illustration Example: To show more clearly how *quantifying path similarity* and *recommending security rules* are conducted, we give an illustration here. For the sake of clarity, we use 3 samples consisted of relatively simple paths and some irrelevant vertices are left out. Note that this is only for illustration purpose and SARRE is able to analyze more complex samples in an unsupervised way, as we will show later in Section V. We set the number of neighbors chosen when making recommendation as 1 here instead of 5, which is commonly adopted in our implementation and given more discussions in Section V-C. The sample packages' names and their event graphs are depicted in Fig. 5. The weights assigned through optimization formulation (5) are considered as given here and shown at the end of vertices. Identified paths are labeled by dotted lines. Package 1 is assigned 'Social', package 2 and 3 are assigned 'Information Retrieval' as *DF* labels. Suppose $P_{2,1}$ is the one requires rule recommendation. First, the packages which share at least one common *DF* label with package 2 are chosen (package 3 in this case), the similarity score between any reference path in the chosen packages and $P_{2,1}$ is calculated by Equation (3). The $P_{3,1}$ is selected as the 1 nearest neighbor with similarity score 0.6667 to $P_{2,1}$. The recommended rule for $P_{2,1}$ is calculated by Equation (4), which gives result 0.2. Thus, a security action corresponding to 0.2 is enforced for the information flow on this event path. Mapping from this number 0.2 to security action is discussed in the next subsection. Note that, in real usage scenarios involving many packages and paths, all calculations are not trivial as current illustration example. Also note that for a path P_r needs recommendation, even if some similar paths exist in packages which share no common *DF* label with the parent app of P_r, their rules will not serve as references for recommendation, because of their mismatched semantic information.

D. Camouflage Engine

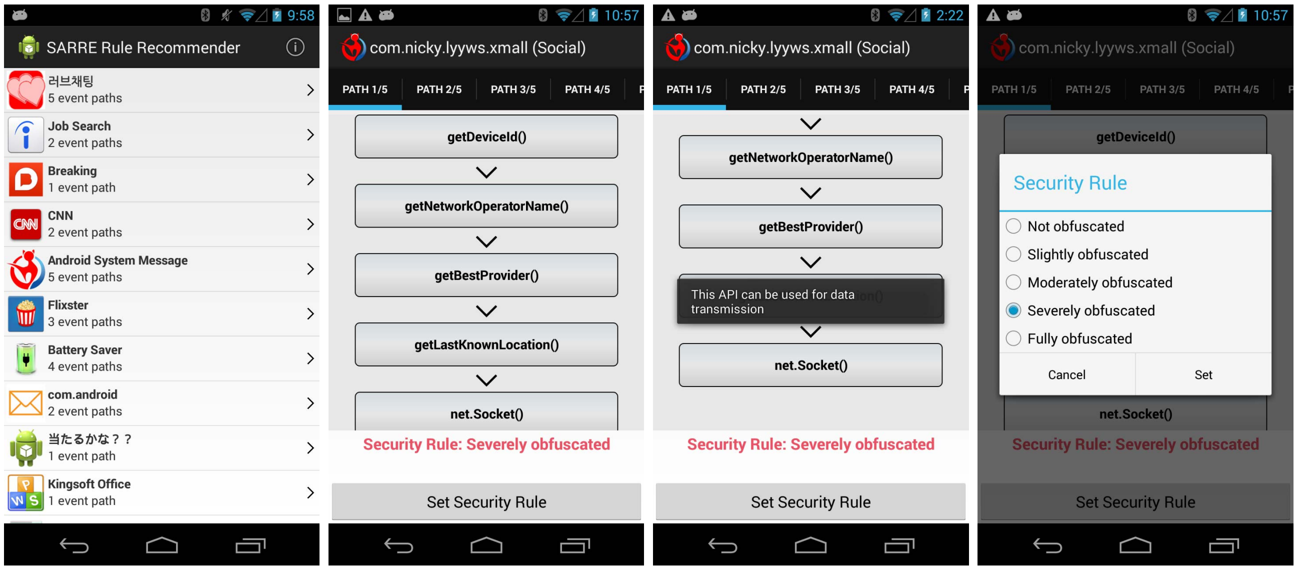
For a given security rule $R_r (0 \leq R_r \leq 1)$ for path P_r, an appropriate camouflage action that depends on the underlying data types is selected to enforce the security rule and obfuscate information flow at run time. Our *Camouflage Engine* defines and enforces three families of camouflage actions for (1) numerical data, (2) string data, and (3) structured data.

1) Camouflaging Numerical Data: For numerical data such as GPS coordinates provided by *LocationManager*, we replace the original data X by a noise-corrupted version $\pi X + (1 - \pi)Y$, where Y is randomly generated noise and π is a proper obfuscation level determined by the security rule (e.g., $R_r = 0.8$ implies location obfuscation at street level, $R_r = 0.2$ implies obfuscation at city level). The mapping between π and security rule R_r is calibrated for each data source.

2) Camouflaging String Data: For a regular string or static data such as *DeviceID* and *phone number*, we map each security rule R_r to partial scrambling of certain characters and digits. For the situations where even partial revelation causes information leakage (e.g., registered operator name can be inferred by a few letters of the value returned by *getNetworkOperatorName*), we only enforce a binary security rule $R_r \in \{0, 1\}$.

3) Camouflaging Structured Data: Some data are accessed using structures containing multiple data fields. For example, using the *query* method to access contacts data, the returned cursor can be used to access different fields of the entry such as first name, last name, and email address. We map each security rule R_r to the obfuscation of certain fields in the data structure, e.g., only first name is visible when $R_r = 0.2$, while first name and email are both shown when $R_r = 0.4$. This mapping is defined for different types of structured data.

Note that in the example of Fig. 5, we depict different events with same API name (e.g. *getLine1Number* in package 3) as the same vertex because of space limitation. In reality, we can further distinguish events with the same API name, but called in different locations in the application, or with different parameters, as discussed above in the end of Section IV-A. In this way, SARRE is able to enforce different rules to the same type of sensitve data when they are associated with different event paths or under different contexts. In rare cases, it is still possible that same type of sensitive data is requested in the same context and with same set of parameters, but lead to different event paths and thus are assigned with different rules. During runtime execution, if such case happens, the most



This API can be used for data transmission

Security Rule

Not obfuscated
Slightly obfuscated
Moderately obfuscated
[✓ Severely obfuscated]
Fully obfuscated

Fig. 6. Screenshots of SARRE's GUI.

strict rule is enforced in our implementation for the sensitive data on the event path.

4) SARRE's GUI: SARRE on Android system together with GUI are shown as Fig. 6. The GUI first shows a list of apps and their summary of event paths managed by SARRE. Selecting any app gives users access to its event paths with current security rules. We map the numerical rules in our implementation to textual descriptions understandable to users. Clicking on an event node on a path gives brief description to the event's meaning. The interface allows user to overwrite security rules set by *Rule Recommender*. Since user inputs reflect their unique preferences, once a modification is made, the updated rule and their associated path will be added to the *Reference Rule DB*. *Rule Recommender* will automatically re-compute security rules of other paths in the system based on the updated information and inference of user preference through weight optimization (5).

Such option reserved for users to overwrite recommended rules aims to give users the flexibility to adjust based on their personal preferences. However, our system doesn't rely on users' rule configurations to be functional. The *Reference Rule DB* in Analysis Server depicted in Fig. 2 is initialized with a rule database obtained from large training set, which enables the SARRE system to be fully functional from the beginning. Existence of noise from users' configured rules is a common problem for recommendation-based solutions. Existing works such as [24] solve this problem by detecting both noise originated from imperfect users' configurations and noise deliberately attempted to bias the recommendation results from malicious users. Besides, exiting works [25], [26] demonstrate that a crowdsourcing-based recommendation system is effective to benefit the 'normal' users from the decisions made by the 'expert' users. Our system design involving a centralized Analysis Server is easily extensible to leverage such techniques, which are complementary with our work.

In current implementation, rarely, similar reference paths may be unavailable for a newly-identified path that needs rule recommendation. This is recognized as code-start problem [27], which is common in recommendation systems. SARRE will select a relatively strict rule for such case, at the same time, it reserves the options for users to overwrite the assigned rules through GUI in Fig. 6. Besides, currently we measure the similarity between event paths strictly based on the events' numerical indexes. One way to further reduce the chance of missing reference paths is to extend the recommendation to be aware of the content of the monitored events. For example, if one user prefers strict rules on his/her contact list, it is likely similar preference exists for call log information. Such content-based approach is proved to be effective in addressing cold-start problem [27] in movie recommendation system utilizing cast information of the movies, and we leave such extension as future work.

V. Evaluation

In this section, we evaluate SARRE from four perspectives: *(i)* the accuracy of path identification, *(ii)* effectiveness of recommended rules to meet users' expectations, *(iii)* the effectiveness of the enforced rules to prevent information leakage, and *(iv)* the runtime system overhead introduced by SARRE.

A. Experimental Setup

1) Test Bed Setup: Our evaluation is on Android devices *Galaxy Nexus*, powered with dual-core 1.2GHz cortex-A9 processor, 1GB memory and 16GB storage. We implemented our design on AOSP (*Android Open Source Project*) v4.1.2. We collected and tested 1706 app samples, including 233 working malware samples collected from an online malware sharing site [21], and the other samples from the top ranking apps on Google Play spanning multiple categories.

2) Testing Tools: For sample testing, we leverage the automated testing tool Monkey [28] for the experiments. For each app, we first use Apktool [29] to extract the package name and main activity name and input them to Monkey. Monkey then launches the app, and continuously generates at least 500 random user events such as clicks, gestures, as well as system-level events. Although a common problem to dynamic testing is the difficulty to achieve exhaustive code coverage, 500 of events injection is confirmed in prior work [30] to provide effective coverage in identifying sensitive behaviors. Also, we use the same seed number [28] to generate random events when testing the same app sample for different groups of experiment.

3) Evaluation Plan: Since no existing benchmarks or automated tools, to the best of our knowledge, can be used to evaluate runtime event path identification accuracy, we resort to both manual code review and TaintDroid [3] (a runtime tainting tool to detect sensitive data leakage) for evaluation of the accuracy of path identification. Also, there is no existing work capable of assigning fine-grained rules in the granularity of event path. To serve as baselines for evaluation of the rule recommendation effectiveness, two methods are derived from the literature for comparison with our method: (i) *Permission-based method* [23] utilizes permissions requested by apps to score apps' security risks. (ii) *API call-based method* [20], [25] utilizes runtime behaviors for sample classification and risk evaluation. Although none of these methods makes a step to security rule assignment or enforcement, we implement two recommendation methods based on (i) apps' permission requests and (ii) runtime API calls, respectively. To guarantee fairness, analogical to what we do in our methods, we attach different weights to permissions or API calls to reflect their relative importance, we also utilize similar optimization as in our method to find the optimal weight assignments for these methods to minimize their errors.

Several feasible ways exist to obtain the *DF* labels including reusing category information provided by app markets, natural language processing tools such as AutoCog [17] and WHYPER [19], and even user selections. In current evaluations, for apps collected from Google Play, we reuse the category information but map apps within several categories such as Weather and News to the same *DF* label, which is Info. (Information Retrieval), because their functionalities are similar, i.e. to retrieve certain types of information from the Internet. For other samples, we currently manually assign one of the following 6 *DF* labels based on app descriptions: (1) Games, (2) Info., (3) Social, (4) Communication (abbreviated as Comm), (5) Tools and (6) Tracking/Maps (abbreviated as Maps). Extending to more *DF* labels can be easily achieved.

B. Accuracy of Path Identification

Using the testing methods discussed above, we test all the 1706 app samples, and analyze the results obtained from *Path Identifier*. We first show the distribution of number of paths and length of paths identified for a single sample. As shown in Fig. 7, the length of paths identified for a single app ranges from 2 to 7, while the number of paths range from 1 to 6. Besides, more than 90% of event paths are longer than 2 events. Investigation of the identified paths shows that this is because besides of the sensitive data access and transmission events, our event paths can also capture other critical events such as system events that serve as triggers for an event path. Also, some event paths involve more than one types of sensitive data, this will happen when sensitive data is sent in a batched manner such as the fifth example *Faketimer* in Table III. For a single app, different event paths can be associated with the same type of sensitive data, but triggered by different events in different locations of the source code, or triggered with different parameters. Such information is captured by *Event Monitor* of SARRE and facilitates context-aware rule enforcement.

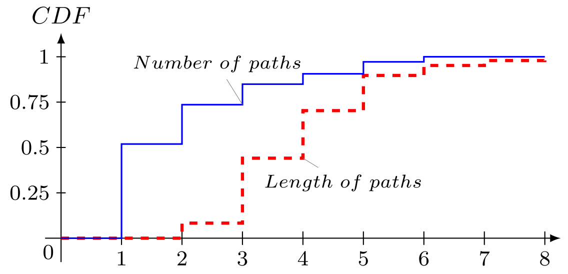


Fig. 7. Cumulative distribution of number of paths and length of paths identified for a single app.

Next, we evaluate the accuracy of identified event paths. We conduct two set of experiments: *(i)* We manually review the code of randomly selected 13 malware samples and 10 popular apps from Google Play. We test the accuracy of *Path Identifier* by comparing the identified paths with code review results. We verify whether the paths identified by our system are true paths existed in the apps, also whether all the critical paths in the apps are correctly identified or not. *(ii)* We test the whole corpus of 1706 samples, including 233 malware samples on devices configured with TaintDroid [3], and verify the sensitive information flows associated with our identified event paths with the results obtained from TaintDroid.

In the first experiment, *the code review and experiments with our system are carried out by two different groups of researchers independently*. One group summarizes the results outputted by our *Path Identifier*, while the other group searches for critical event paths only by reviewing the packages' source code. No results are exchanged until the final phrase, when the results from two groups are compared. We believe such isolation ensures the evaluation's objectiveness. The results show that 43 information flow paths are identified by *Path Identifier*, while code review of these samples gives 42 information flow paths. Comparison of these two set of results shows that among 42 paths given by code review, 40 of them are covered by those identified by *Path Identifier*. Among the 43 paths identified by *Path Identifier*, only 3 of them cannot be verified by code review. The summary of sample paths and the explanations of accuracy evaluation based on code review are listed in the second and third columns in Table III, respectively.

As noted in [3], TaintDroid does not support apps which include native libraries and there is a growth in the number of apps relying on native libraries. Although the ratio of apps included a *.so* file is as low as 5% in 2010, we find it reaches 53.75% based on analysis of the 1706 samples we collected.

TABLE III
SAMPLE PATHS IDENTIFIED (FIRST 13 SAMPLES ARE MALWARE AND OTHERS ARE FROM GOOGLE PLAY)

Sample	**Sensitive information involved on identified paths**	**Comments**
1. Nickispy	(i) telephony†, location, (ii) SMS, (iii) location, (iv) call record	+ All paths match
2. GGtracker	(i) telephony, (ii) (iii) SMS ((ii) triggered by new SMS, (iii) by *HomeActivity* onStop), (iv) sending SMS	− Path (iv) is not verified by code review
3. Fakedaum	(i) Line1Number, SIM info., (ii) SMS, (iii) contacts	+ All paths match
4. Systemsecurity	(i) device ID, subscriber ID, (ii) contacts	+ All paths match
5. Faketimer	(i) accounts, telephony, location, (ii) telephony, location, (iii) location	+ All paths match
6. Faketaobao	(i) telephony	− 1 path for payload retrieval is missing
7. Loozfon	(i) telephony, contacts	+ All paths match
8. LoveChat	(i) telephony, contacts, (ii) SMS	+ All paths match
9. SMSspy	(i) telephony, (ii) SMS	+ All paths match
10. Exprespam	(i) telephony	− 1 path of contact info. is missing, since malware's C&C server is down.
11. Dougalek	(i) telephony, contacts	+ All paths match
12. Spamsoldier	(i) telephony, contacts	− missed SMS spamming path, since malware's C&C server is down.
13. Threatjapan	(i) contacts	+ All paths match
14. Accuweather	(i) (ii) (iii) location, (iv) device ID, (v) device model	− (iii) is not verified by code review
15. Jobsearch	(i) location, (ii) device ID	+ All paths match
16. Flixster	(i) device ID, model info, (ii) device ID, (iii) location	+ All paths match
17. CNN mobile	(i) carrier info., device ID, device model, (ii) location	+ All paths match
18. Angry Birds	(i) device model and ID, (ii) network operator, device ID	+ All paths match
19. Zentertain	(i) device ID	+ All paths match
20. BreakingNews	(i) device name, manufacturer info	+ All paths match
21. Wpsoffice	No information flow paths are identified	+ Match code review
22. Game2048	No information flow paths are identified	+ Match code review
23. Call Log	No information flow paths are identified	+ Match code review

† 'telephony' is short for telephony service related information in the table
− denotes mismatch between paths identified by 'SARRE' and code review, and explanations. Note that other paths in the same app except these mismatches are exact matches
+ denotes that for the sample, not only paths obtained by code review are correctly identified (no false negatives), but also paths not obtained by code review are not falsely identified (no false positives)

As a result, 789 of app samples are supported by TaintDroid including 197 malware samples. The apps collected from Google Play supported by TaintDroid include 131 Information Retrieval apps, 40 Games, 57 Communication apps, 57 Social apps, 265 Tools, 53 Tracking/Maps apps. Comparing the results obtained by *Event Identifier* of SARRE and TaintDroid, in most cases, the information flows identified by SARRE match those leakage alerts given by TaintDroid. For each category, we plot the number of matched event paths between TaintDroid and SARRE and those identified by SARRE or TaintDroid only in Fig. 8. We use the results of TaintDroid as TP (True Positive), those information flows identified by SARRE but not verified by TaintDroid's results as FP (False Positive), and those alerted by TaintDroid but not reported by SARRE as FN (False Negative). Calculating the evaluation metrics $Precision = \frac{TP}{TP+FP}$ and $Recall = \frac{TP}{TP+FN}$, our method overall achieves 93.8% Precision and 96.4% Recall.

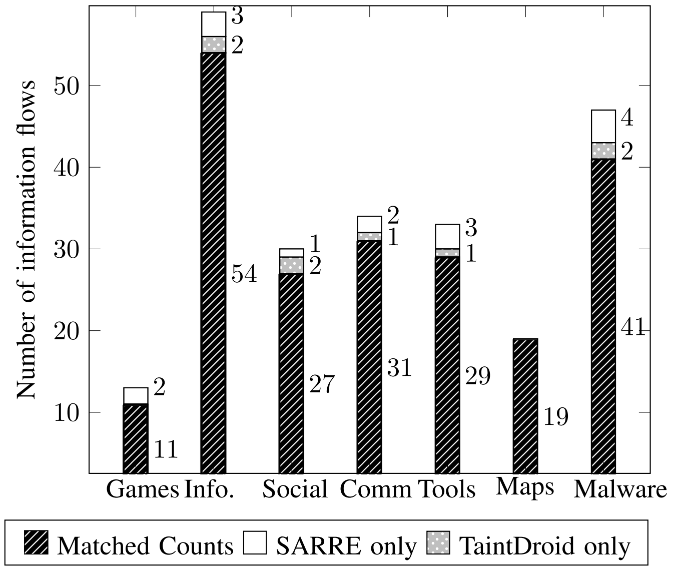


Fig. 8. Results comparisons of information flows by SARRE with leakage alerts by TaintDroid.

C. Effectiveness of Rule Recommendation and Enforcement

From the app samples tested, we randomly choose 67 benign samples and 8 malware samples to build our testing set, while the rest as training set. This size is chosen because numbers of apps used by general users normally range from 10 to 90 [31]. Before evaluation, we assign one of the six *DF* labels to each sample, and pre-configure the rules (within range $[0, 1]$) for all paths from a user's perspective. We use MAE (*Mean Absolute Error*), a widely used accuracy

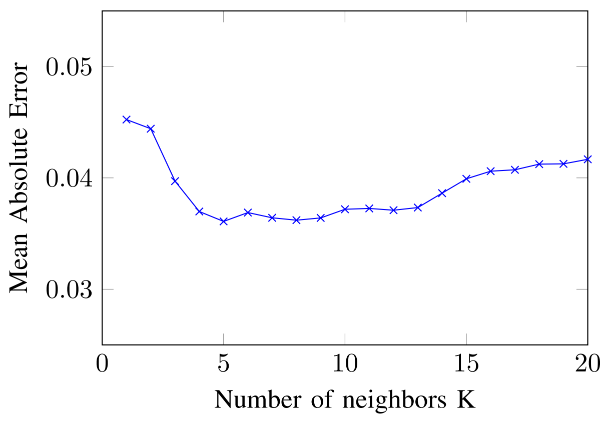


Fig. 9. Variations of our method's MAE between recommended rules and pre-configured rules, when K is increased from 1 to 20.

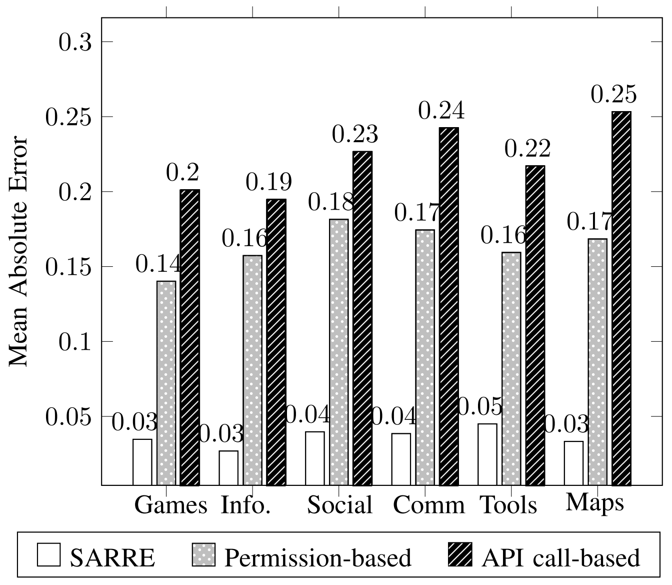


Fig. 10. Comparisons of our method's MAE with two other methods for each testing set.

evaluation metric [14], [32] for recommendation systems. The MAE is calculated by averaging the accumulated absolute difference between pre-configured and recommended rules over all testing paths.

1) How Many Neighbors Are Enough?: With the number of nearest neighbor K increases from 1 to 20, we evaluate the MAE between recommended and pre-configured rules, as depicted in Fig. 9. When K is smaller than 5, accuracy is improved as increasing number of reference paths become available, however, when K is larger than 8, more noise is introduced by reference paths which are less similar to the path that requires rule recommendation. Similar shape of error has also been observed by former work of neighbor-based prediction [14], [32]. In our work, we fix the K as 5.

2) How Accurate Is the Rule Recommendation?: To evaluate SARRE's *Rule Recommender* against the two methods we discussed in Section V-A, we plot the MAE comparison for each testing set consisted of apps with the same *DF* label, as seen in Fig. 10. The MAE measures how effective

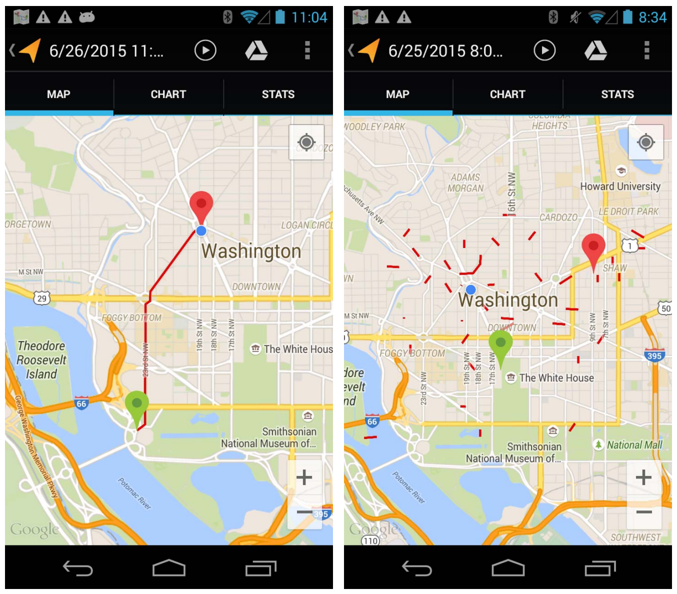


Fig. 11. Tracks when different rules are enforced (left: '1' is enforced, the track is accurate, right: '0.4' is enforced, the track is with noise).

the method is to make recommendations to meet the users' expectations of security rule assignment. We can see in each scenario, our recommendation method in SARRE outperforms the methods based on static requested permissions or runtime API call statistics. The evaluation result demonstrates the necessity of event paths and semantics-awareness in security rule recommendation.

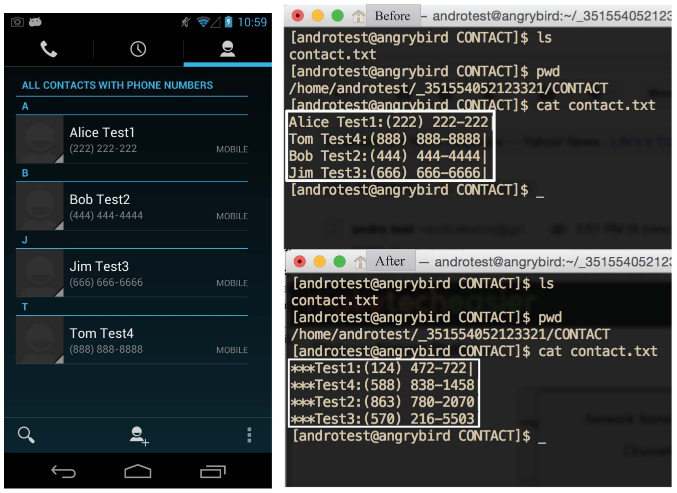
3) How Effective Is the Rule Enforcement?: We illustrate the effectiveness of the rule enforcement by two examples we introduced as motivational examples in Section II.

a) My tracks [15]: Since malware normally doesn't present harvested data when stealthily eavesdropping on users' location, we use a tracking app *My Tracks* to emulate malware by intentionally replacing its actual *DF* label 'Tracking/Maps' by 'Games'. We choose a tracking app because it has a UI showing GPS coordinates update and makes it convenient to compare the data when different rules are enforced. An event path identified for *My Tracks* involving location data is written in an abbreviated manner, as follows:

$$\boxed{\textit{GPS updated} \rightarrow \textit{getLocation} \rightarrow \textit{Socket.getOutputStream} \rightarrow \textit{Socket.connect}}$$

The recommended rule for this path, after rounding up to discrete level is 0.4, which means a security action corresponding to 0.4 needs to be applied when it shows up in a game app. When we replace the true *DF* label 'Tracking/Maps' back to this app, the rule recommended and enforced is 1, which means such an event path in a 'Tracking/Maps' app should be left intact. The tracks with rules 1 and 0.4 enforced are shown in Fig. 11. Malware such as *Nickispy* [16] and *Faketimer* eavesdropping users' location exhibit similar event paths as the event path above, and after rule enforcement the location data sent should be similar to the right one in Fig. 11.

b) Love chat [18]: This malware with package name *com.yxx.jiejie* doesn't show an UI. It has an event path


ALL CONTACTS WITH PHONE NUMBERS

A
Alice Test1 (222) 222-222 MOBILE

B
Bob Test2 (444) 444-4444 MOBILE

J
Jim Test3 (666) 666-6666 MOBILE

T
Tom Test4 (888) 888-8888 MOBILE

Before – androtest@angrybird:~/_35155405212

```
[androtest@angrybird CONTACT]$ ls
contact.txt
[androtest@angrybird CONTACT]$ pwd
/home/androtest/_351554052123321/CONTACT
[androtest@angrybird CONTACT]$ cat contact.txt
Alice Test1:(222) 222-222|
Tom Test4:(888) 888-8888|
Bob Test2:(444) 444-4444|
Jim Test3:(666) 666-6666|
[androtest@angrybird CONTACT]$ _
```

After – androtest@angrybird:~/_35155405212

```
[androtest@angrybird CONTACT]$ ls
contact.txt
[androtest@angrybird CONTACT]$ pwd
/home/androtest/_351554052123321/CONTACT
[androtest@angrybird CONTACT]$ cat contact.txt
***Test1:(124) 472-722|
***Test4:(588) 838-1458
***Test2:(863) 780-2070
***Test3:(570) 216-5503
[androtest@angrybird CONTACT]$ _
```

Fig. 12. Contacts data on the phone (left) and data sent by malware *Love Chat* when rule '1' (upper right) or '0.2' (lower right) is enforced.

identified as follows, in which data is accessed, stored locally and sent out by network socket.

getLine1Number → *getDeviceId* → *query(contacts)* → *io.FileOutputStream* → *Socket.getOutputStream*

Based on this sample's declared functionality, we attach 'Communication' as its *DF* label. The recommended rule by our *Rule Recommender* for this path is 0.2. With examination of the *Reference Rule DB*, we see that although this malware is disguised as a 'Communication' app, it doesn't get a rule with large number, because paths similar to the above path are popular among privacy-stealing malware, but not 'Communication' apps. This makes sense and shows the necessity to use event path as reference for rule recommendation. Since this app sends harvested information on the background, to see the enforcement effect, we redirect its sent packets to a server we deployed. We input some made-up contacts data on the phone as shown in Fig. 12 (left). The contents in the files it sent to the emulated server before and after rule enforcement are also shown in the figure (right). We can see the effectiveness of the rule enforcement by hiding contacts' first names, scrabbling some digits in the phone numbers.

D. System Performance Evaluation

To measure SARRE's overhead on real devices, similar to the approach used in prior works [6], [33], we use a popular Android system benchmark *softweg* [34]. We ran *softweg* 20 times on devices installed with AOSP (*Android Open Source Project*), and AOSP with SARRE implemented. The performance differences are listed in Table IV. 'SD' and 'RSD' refer to scores' standard deviation and relative standard deviation.

The 'mean' for micro-benchmarks of creating and deleting empty files represents time consumed (the lower, the better), while all other metrics represent scores (the higher, the better). SARRE implementation has no direct effects on graphics performance, and the total graphics overhead measured with transparent and opaque image overlays is only 0.02%.

TABLE IV
SARRE PERFORMANCE EVALUATION

	AOSP with SARRE			AOSP			Overhead
	mean	SD	RSD	mean	SD	RSD	
graphics	**19.46**	**0.34**	**1.73%**	**19.46**	**0.32**	**1.65%**	**0.02%**
opacity	6.32	0.14	2.21%	6.27	0.20	3.15%	-0.66%
transparent	5.59	0.11	1.90%	5.63	0.12	2.07%	0.78%
CPU	**3501.5**	**80.07**	**2.29%**	**3534.0**	**75.07**	**2.12%**	**0.92%**
mwips dp	206.59	15.03	7.27%	213.21	6.98	3.28%	3.10%
mwips sp	313.31	7.17	2.29%	310.76	7.94	2.55%	-0.82%
mflops dp	26.07	3.13	12.02%	27.55	2.16	7.86%	5.37%
mflops sp	63.72	8.20	12.86%	68.08	3.24	4.76%	6.40%
vax mips dp	142.94	3.87	2.70%	145.39	4.46	3.07%	1.68%
vax mips sp	220.24	2.70	1.23%	216.95	4.53	2.09%	-1.52%
	AOSP with SARRE			**AOSP**			**Overhead**
	mean	**SD**	**RSD**	**mean**	**SD**	**RSD**	
memory	**492.22**	**66.22**	**13.45%**	**543.43**	**57.70**	**10.62%**	**9.42%**
copy memory	447.27	60.17	13.45%	493.80	52.44	10.62%	9.42%
file system	**200.29**	**7.38**	**3.69%**	**231.54**	**6.14**	**2.65%**	**13.50%**
create 1000 files	1.05	0.09	8.11%	0.37	0.02	6.38%	184.41%
delete 1000 files	0.26	0.03	10.04%	0.25	0.02	9.63%	2.45%
write to FS	87.45	4.84	5.53%	97.83	3.44	3.52%	10.61%
read from FS	315.51	11.55	3.66%	367.99	10.50	2.85%	14.26%
create 250 files	2.60	0.17	6.60%	1.45	0.09	5.93%	79.26%
SDcard write	36.96	1.56	4.21%	39.84	1.73	4.35%	7.22%
SDcard read	68.98	2.74	3.98%	73.46	2.82	3.83%	6.10%

For CPU performance, in total only 0.92% overheads are introduced by SARRE. The CPU overhead are evaluated in terms of various operations, such as *mwips dp* and *mwips sp* (Millions of Whetstone Instructions Per Second Double Precision and Single Precision). Note that for micro-benchmarks *opacity graphics, mwips sp* and *vax mips sp*, the overheads measured are negative, but their absolute values are much smaller compared with the RSD of the 20 times of experiments. They are considered as not significant, and originates from experiment variance, rather than performance improvements introduced by SARRE. Such variances are observed in related work [6] using the same set of benchmarks. For memory, the overhead is 9.42%, which is also acceptable. For file system I/Os, SARRE seems to introduce 13.50% overhead, which is mostly due to the 'creating 1000 empty files' and 'creating 250 empty files' benchmarks, where overheads are 184.41% and 79.26%. However, continuously creating such a large number of files without other operations is rare in real usage scenarios, the latency will be amortized over time and should exhibit smaller impact on users' phones. Also note that in our current evaluation, file encryption and transmission happens in every 10 minutes. In real cases, users normally interact with the apps and generate log files in much lower density [31]. We foresee simple optimization works such as transmission scheduling can further reduce the overhead.

VI. Related Work

A. Information Flow Analysis

Prior work detects information leakage based on flow tracking [3], statistical analysis on source and sink data similarity [35], static code analysis [11], combination of code analysis and runtime event tracking [36]. Apart from identifying data source and sink pairs as these methods do, our method also builds the full scenarios of the event paths including triggering events, and gives a runtime solution to take finergrained security actions taming information leakage. Another direction of work related to our recommendation system is the detection of unjustified information transmission based on user inputs [37], user intents [38], system state changes [39], or peer voting [40]. However, these works serve different goals such as malware detection [37], assistive or automatic analysis [38]–[40]. Our work makes a step to runtime rule assignment to each extracted path.

B. Runtime Testing and Monitoring

Runtime app testing techniques are proposed to analyze permission leaks [41], [42]. AppsPlayground [43] uses both fuzzing and tainting methods. DroidScope [33] utilizes binary instrumentation for runtime testing. While these works provide efficient ways for app analysis, they normally pay limited attentions to automatic analysis and utilization of testing results. They complements SARRE in that these works enable us to fully test more apps. On the other hand, SARRE extends runtime monitoring to automatic rule assignment and enforcement after testing results become available.

C. Malware Detection

Malware detection has been extensively studied employing features like permissions [23], API calls [20], [25], API dependency graph [44], behavioral graph [45] or combination of several features [46]. Because of space limitation, we are not able to list all prior works. SARRE serves a goal orthogonal to malware detection and it applies to not only malware, but all apps in Android system.

D. Security Enhancement

Static code analysis methods [10], [11] for automated security specification inference are not applicable to Android apps, which are developed with extensive code encryption, obfuscation and dynamically loaded code. Our work is complementary with prior work on contextual access control such as Pegasus [47], SE Android [6], *SEACAT* [5], MobileIFC [48] and ConUCON [49], data obfuscation techniques in MockDroid [8], TISSA [9], security profiles enforcement and switching in MOSES [50], context-related policy enforcement in CRêPE [51], and fine-grained sensor management in SemaDroid [52]. Most of these works heavily rely on developers or users for convoluted rule configurations, which can be mitigated by our rule recommendation system. On the other hand, these works can help to conveniently broaden mediated resource list in our work with their policy language support or resource management techniques.

We have presented some preliminary ideas of rule recommendation and enforcement as well as two proof-of-concept case studies in a previous poster [53]. However, this journal submission includes substantial new contents: details about system design, algorithms on event path identification and rule recommendation based on path similarity measurement, complete system implementation, and new experiment results including evaluation of path identification accuracy and rule recommendation effectiveness, quantitative comparisons with other approaches and evaluation of system performance overhead.

VII. Conclusion and Future Work

We propose SARRE, a novel solution to provide fine-grained, semantics-aware protection over users' private data. SARRE leverages statistical analysis and a novel application of minimum path cover algorithm to identify system runtime event paths. Then it automatically generates security rules for information flows on different paths based on both known rules of similar paths and semantic information. The SARRE system is prototyped on Android devices and evaluated with 1473 popular apps from Google Play spanning multiple categories, and 233 real-world malware samples. The results show that SARRE overall achieves 93.8% precision and 96.4% recall in identifying the event paths, and also the average relative error between rule recommendation and manual configuration is less than 5%, validating the effectiveness of automatic rule recommendation. A camouflage engine implemented to enforce the recommended rules demonstrates SARRE is effective to provide the fine-grained protection over private data with low performance overhead.

Future Work: Although SARRE is transparent to the installed apps in the system, it is possible for a determined attacker to detect the existence of SARRE and launch specially designed attack procedures to evade SARRE. New hardware-based techniques such as TrustZone adopted by existing work [54] can be used to protect critical system services from determined attackers. It is possible to implement SARRE in this way to further protect SARRE itself.

To evade detection, malware may store the harvested data in memory or file system, and later send the data using network sockets. Although in all the samples we tested, such behaviors are captured by SARRE, it is possible that if the malware intentionally prolong the delay to very long time, there is a chance for SARRE to miss such a link. This problem is solvable by adopting a lightweight version of file operation correlation technique [55] to further link the disjointed two paths to become one event path. For our purpose, it is effective enough to only analyze those files involved in the event paths identified by SARRE, hence, the extra overhead is expected to be much lower than 9%, as reported in [55]. We leave such extension as future work.

References

[1] *Android Still Triggers the Most Mobile Malware*, accessed in 2014. [Online]. Available: http://goo.gl/FXTGsi

[2] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 95–109.

[3] W. Enck *et al.*, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, p. 5, Jun. 2014.

[4] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. CCS*, 2011, pp. 627–638.

[5] S. Demetriou *et al.*, "What's in your dongle and bank account? Mandatory and discretionary protection of Android external resources," in *Proc. NDSS*, 2015, pp. 1–15.

[6] S. Smalley and R. Craig, "Security enhanced (SE) Android: Bringing flexible MAC to Android," in *Proc. NDSS*, 2013, pp. 1–18.

[7] M. Zhang and H. Yin, "Efficient, context-aware privacy leakage confinement for Android applications without firmware modding," in *Proc. ASIACCS*, 2014, pp. 259–270.

[8] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "MockDroid: Trading privacy for application functionality on smartphones," in *Proc. HotMobile*, 2011, pp. 49–54.

[9] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on Android)," in *Proc. TRUST*, 2011, pp. 93–107.

[10] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "AutoISES: Automatically inferring security specifications and detecting violations," in *Proc. USENIX Secur. Symp.*, 2008, pp. 379–394.

[11] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: Specification inference for explicit information flow problems," in *Proc. SIGPLAN PLDI*, 2009, pp. 75–86.

[12] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. Srivastava, "ipShield: A framework for enforcing context-aware privacy," in *Proc. NSDI*, 2014, pp. 143–156.

[13] S. C. Ntafos and S. L. Hakimi, "On path cover problems in digraphs and applications to program testing," *IEEE Trans. Softw. Eng.*, vol. 5, no. 5, pp. 520–529, Sep. 1979.

[14] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proc. WWW*, 2001, pp. 285–295.

[15] *App My Tracks*, accessed in 2015. [Online]. Available: https://goo.gl/7cCbIH

[16] Malware With Package Name *com.nicky.lyyws.xmall*, accessed in 2014. [Online]. Available: http://goo.gl/U7D2FW

[17] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "AutoCog: Measuring the description-to-permission fidelity in Android applications," in *Proc. CCS*, 2014, pp. 1354–1365.

[18] *Mobile Malware Sharing*. [Online]. Available: http://goo.gl/YNIOLg

[19] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *Proc. USENIX Conf. Secur.*, 2013, pp. 527–542.

[20] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in Android," in *Proc. SecureComm*, 2013, pp. 86–103.

[21] *Mobile Malware Sharing Site*, accessed in 2014. [Online]. Available: http://goo.gl/YNIOLg

[22] R. Agrawal *et al.*, "Fast algorithms for mining association rules in large databases," in *Proc. VLDB*, 1994, pp. 487–499.

[23] H. Peng *et al.*, "Using probabilistic generative models for ranking risks of Android apps," in *Proc. CCS*, 2012, pp. 241–252.

[24] M. P. O'Mahony, N. J. Hurley, and G. C. M. Silvestre, "Detecting noise in recommender system databases," in *Proc. IUI*, 2006, pp. 109–115.

[25] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proc. SPSM*, 2011, pp. 15–26.

[26] Y. Agarwal and M. Hall, "ProtectMyPrivacy: Detecting and mitigating privacy leaks on iOS devices using crowdsourcing," in *Proc. MobiSys*, 2013, pp. 97–110.

[27] A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock, "Methods and metrics for cold-start recommendations," in *Proc. SIGIR*, 2002, pp. 253–260.

[28] *UI/Application Exerciser Monkey*, accessed in 2015. [Online]. Available: http://goo.gl/zPMo94

[29] *Reverse Engineering Tool Apktool*, accessed in 2014. [Online]. Available: http://goo.gl/OAvF1e

[30] C. Yang, G. Yang, A. Gehani, V. Yegneswaran, D. Tariq, and G. Gu, "Using provenance patterns to vet sensitive behaviors in Android apps," in *Proc. SecureComm*, 2015, pp. 58–77.

[31] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in smartphone usage," in *Proc. MobiSys*, 2010, pp. 179–194.

[32] J. L. Herlocker, J. A. Konstan, A. Borchers, and J. Riedl, "An algorithmic framework for performing collaborative filtering," in *Proc. SIGIR*, 1999, pp. 230–237.

[33] L.-K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. USENIX Secur. Symp.*, 2012, pp. 230–237.

[34] *Android Benchmark Softweg*, accessed in 2015. [Online]. Available: https://goo.gl/8G3lDm

[35] O. Tripp and J. Rubin, "A Bayesian approach to privacy enforcement in smartphones," in *Proc. USENIX Secur. Symp.*, 2014, pp. 175–190.

[36] J. Yu, S. Zhang, P. Liu, and Z. Li, "LeakProber: A framework for profiling sensitive data leakage paths," in *Proc. CODASPY*, 2011, pp. 75–84.

[37] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu, "pBMDS: A behavior-based malware detection system for cellphone devices," in *Proc. WiSec*, 2010, pp. 37–48.

[38] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection," in *Proc. CCS*, 2013, pp. 1043–1054.

[39] X. Chen and S. Zhu, "DroidJust: Automated functionality-aware privacy leakage analysis for Android applications," in *Proc. WiSec*, 2015, Art. no. 5.

[40] K. Lu *et al.*, "Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting," in *Proc. NDSS*, 2015, pp. 1–15.

[41] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "IntentFuzzer: Detecting capability leaks of Android applications," in *Proc. ASIA CCS*, 2014, pp. 531–536.

[42] Y. Zhang *et al.*, "Vetting undesirable behaviors in Android apps with permission use analysis," in *Proc. CCS*, 2013, pp. 611–622.

[43] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic security analysis of smartphone applications," in *Proc. CODASPY*, 2013, pp. 209–220.

[44] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *Proc. CCS*, 2014, pp. 1105–1116.

[45] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in Android applications," in *Proc. ESORICS*, 2014, pp. 163–182.

[46] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, 2014, pp. 1–15.

[47] K. Z. Chen *et al.*, "Contextual policy enforcement in Android applications with permission event graphs," in *Proc. NDSS*, 2013.

[48] K. Singh, "Practical context-aware permission control for hybrid mobile applications," in *Research in Attacks, Intrusions, and Defenses*. Saint Lucia: Rodney Bay, 2013.

[49] G. Bai, L. Gu, T. Feng, Y. Guo, and X. Chen, "Context-aware usage control for Android," in *Proc. SecureComm*, 2010, pp. 326–343.

[50] Y. Zhauniarovich, G. Russello, M. Conti, B. Crispo, and E. Fernandes, "MOSES: Supporting and enforcing security profiles on smartphones," *IEEE Trans. Dependable Secure Comput.*, vol. 11, no. 3, pp. 211–223, May/Jun. 2014.

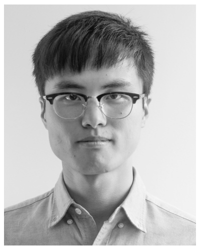
[51] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "CRêPE: A system for enforcing fine-grained context-related policies on Android," *IEEE Trans. Inf. Forensics Security*, vol. 7, no. 5, pp. 1426–1438, Oct. 2012.

[52] Z. Xu and S. Zhu, "SemaDroid: A privacy-aware sensor management framework for smartphones," in *Proc. CODASPY*, 2015, pp. 61–72.

[53] Y. Li, F. Yao, T. Lan, and G. Venkataramani, "POSTER: Semantics-aware rule recommendation and enforcement for event paths," in *Proc. SecureComm*, 2015, pp. 572–576.

[54] W. Li, H. Li, H. Chen, and Y. Xia, "AdAttester: Secure online mobile advertisement attestation using trustzone," in *Proc. MobiSys*, 2015, pp. 75–88.

[55] S. T. King and P. M. Chen, "Backtracking intrusions," *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 223–236, 2005.



Yongbo Li is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, The George Washington University. His research interests include system security, mobile energy accounting, and energy management.

Fan Yao is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, The George Washington University. His research interests are system security and data center energy optimization.

Tian Lan received the B.S. degree in electrical engineering from Tsinghua University in 2003, the M.S. degree from the Department of Electrical and Computer Engineering, University of Toronto, in 2005, and the Ph.D. degree from the Department of Electrical Engineering, Princeton University, in 2010. He joined the Department of Electrical and Computer Engineering, The George Washington University, in 2010, where he is currently an Associate Professor. His research interests include mobile energy accounting, cloud computing, and cyber security. He received the best paper award from the IEEE Signal Processing Society 2008, the IEEE GLOBECOM 2009, and the IEEE INFOCOM 2012.

Guru Venkataramani (SM'15) received the Ph.D. degree from the Georgia Institute of Technology, Atlanta, in 2009. He has been an Associate Professor of Electrical and Computer Engineering with The George Washington University since 2009. His research area is computer architecture, and his current interests are hardware support for energy/power optimization, debugging, and security. He was a recipient of the NSF Faculty Early Career Award in 2012, the Best Poster Award in IEEE/ACM PACT 2011, and the ORAU Ralph E. Powe Junior Faculty Enhancement Award in 2010. He has an invited article in ACM OS Review as an author of the Best of Hotpower 2011 Workshop. He is a Senior Member of ACM.